Patterns of Safe Collaboration

Alfred Spiessens

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

February 2007

Faculté des Sciences Appliquées
Département d'Ingénierie Informatique
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Patterns of Safe Collaboration

by Alfred Spiessens

Computer Science and Engineering Department
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

Abstract

When practicing secure programming, it is important to understand the restrictive influence programmed entities have on the propagation of authority in a program. To precisely model authority propagation in patterns of interacting entities, we present a new formalism Knowledge Behavior Models (KBM). To describe such patterns, we present a new domain specific declarative language SCOLL (Safe Collaboration Language), which semantics are expressed by means of KBMs.

To calculate the solutions for the safety problems expressed in SCOLL, we have built SCOLLAR: a model checker and solver based on constraint logic programming. SCOLLAR not only indicates whether the safety requirements are guaranteed by the restricted behavior of the relied-upon entities, but also lists the different ways in which their behavior can be restricted to guarantee the safety properties without precluding their required functionality and (re-)usability. How the tool can help programmers to build reliable components that can safely interact with partially or completely untrusted components is shown in elaborate examples.

Acknowledgments

First of all, I want to thank my thesis advisor professor Peter Van Roy for giving me the opportunity to become a PhD student in his group, and for his valuable guidance and enthusiastic support of my research work. I consider myself very lucky to have been part of a research team that was constantly driven by his enthusiasm for good software research, and could always count on his relentless energy to turn bad news into challenging opportunities.

I am grateful to the members of my thesis advice committee for their advice and assistance at different stages of the preparation of my thesis: professor Jean-Jaques Quisquater, professor Peter Van Roy, and Dr. Mark Miller. They all had a decisive influence on the contents of this thesis, as they wisely advised me to include a more formal aspect into my research, advice I came to appreciate a lot.

Without any doubt, Mark Miller was my most committed mentor and ally in the quest for secure software. To him I owe my current understanding of the role and the importance of capabilities for secure software. I am especially grateful for the many occasions on which he flew over to Belgium to work together and give me the opportunity to learn from his tremendous experience in capability based security and language design. The basis for the work presented in this thesis was laid during a three week session, partially spent discussing in the quiet of the Belgian Ardennes.

I want to thank professor Frank Piessens for the interesting discussions we had on secure software and for his encouraging comments, and professor Yves Deville, the chairman of my thesis committee.

To my coworkers in the MILOS project and in the Distoz research group I am grateful for their encouragements and for the many discussions we had on capability based security and software engineering principles and practices. I thank the Walloon Region for funding the MILOS project.

I thank especially Dr. Luis Quesada for the fruitful collaboration which resulted in a common chapter in this thesis and Raphaël Collet for his collaboration and co-authorship on declarative on-demand computation in Mozart/Oz and his guidance in constraint programming.

I thank Yves Jaradin especially for his support and patient advice on the formal parts of my work, for his indispensable contributions to the core ideas of SCOLL, for his experiments with alternative implementations of SCOLLAR, for the versatile parser he built that allowed me to experiment with the syntax of SCOLL, and for his co-authorship in several papers and technical reports, but above all for always being the first to understand my concerns and aspirations about SCOLL.

I also want to mention the direct and indirect support I had from Boris Mejías and Raphaël Collet, my colleagues at UCL, co-founders of the band we appropriately called "confused deputies" in honor of an important challenge for safe collaboration that is modelled and discussed in this work.

I thank the Mozart/Oz hacker and developer communities for their support. I thank the members of the cap-talk mailing list for the many interesting discussions on the list. I also thank the members of the E-language community for keeping the capability based language E alive and well, and for the discussions on the e-lang mailing list.

I thank Dr. Barry O'Sullivan and professor Gene Freuder of Cork Constraint Computation Centre and my coworkers at that research lab for giving me the opportunity to work with them and to continue my research in a very stimulating environment. I am grateful to the Science Foundation Ireland for funding the work I did there.

I also want to mention here the company "objective development", who offered me the free use of their product "Little Snitch" [lit] for the purpose of my research. Their product is discussed as an example of ACL based reference monitoring in Section 8.1.5.

Last but not least, I want to thank two people that were not directly involved in my PhD research, but had a definitive influence on my education and carrier as a software engineer: professor Theo D'Hondt of VUB (Free University of Brussels) and my good friend Michel Tilman of IMEC Leuven, who remain my most important mentors in software engineering, both in theory and practice.

Fred Spiessens
February 21, 2007

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Safety Analysis for Software Engineers

The subject of this thesis is safety analysis in systems of interacting entities, applied to software development and engineering. Together with privacy and integrity, safety is an important aspect of software security. Safety is about what events should never occur, what actions should never be possible, and what situations should never be reached. Because we assume that situations are concepts that can be rich enough to include accurate descriptions of the events that occurred in the past and of the actions that are possible in the present, we restrict our attention to situations.

The situations that can be reached via the active or passive, direct or indirect involvement of a software entity (component, procedure, object), will be called that entity's *authority*. In practice, a protection system (Section 1.3.1) will impose certain restrictions on the authority of the entities and on the ways in which authority can propagate among the entities, by requiring the availability of certain rights or *permissions*. A more precise definition of authority and permissions will follow (Section 1.3.3).

This work proposes a new formal model (Chapter 5), a declarative language (Chapter 6) and a safety analysis tool (Chapter 7), that allow designers and developers of secure software to express and analyze the safety problems that are of concern to them.

The goal is to provide the following abilities to the software engineer:

1. Choose, refine, or construct a model for authority in her software, and for the way authority can propagate, in accordance with the protection system that controls the software at runtime.

2. Safely approximate the possible runtime effects of the software that is being designed (or the existing software that is to be analyzed) in that model: making sure that all effects that are attainable in the software correspond to reachable authority in the model.

3. Express her concerns about the safety of the software in the model: what authority should not be reachable.

4. Express her concerns about the authority that is necessary for the functionality of the software, and that should not be prevented by the way the enforcement of the safety requirements is (or will be) implemented.

5. Use the proposed analysis tool for one or more of the following purposes :
    - To prove that the safety requirements are satisfied in the model (and thus in software), and that the authority that is necessary for the functionality can be implemented without violating the safety requirements.
    - To discover additional requirements for the software, that, when relied upon, will suffice to make the safety requirements provable in the model, without preventing the implementation of the necessary functionality.
    - To find clues about how and where the model of the software should be refined to improve the resolving power of the analysis.
    - To detect and locate the need for a (partial) redesign of the software, involving the tactical inter-positioning of extra relied-upon software entities, and/or the redistribution of permissions.

New about this approach is:

1. The entities' restrictive influence on the reachable authority can play a role of equal importance to the restrictive influence the protection system exerts via its control of permissions.

2. The dynamic character of an entity's involvement (in making authority reachable) is recognized, and can be safely approximated in the model to an appropriate level of precision. Because of its dynamic nature, we will call an entity's involvement its *behavior* (Section 1.3.7).

3. The explicit and separate modeling of entity behavior and protection system restrictions, each at an appropriate level of precision.

4. The possibility to start with a very simple model of the software, and incrementally refine the model in the parts that need to be modeled in greater detail, enhancing the expressive power of the model precisely where and when it is needed.

The main goal of this work is to provide assistance for the *design* of safe software, by encouraging software engineers and programmers to analyze, understand, and use abstract patterns for safe interaction between the relied-upon parts of the software and the untrusted or unknown parts the programmer must rely on for the functionality they provide without making his software vulnerable to their potential lack of safety.

Nevertheless, the approach presented here can also be used to analyze the safety in existing software. Some examples will illustrate the process of modeling existing source code, to provide a better intuition about the meaning and the role of the constituents in the model, but the scope of this thesis does *not* include the design or implementation of tools for:

- Semi-automatic modeling of existing software from the source code, for the purpose of automated safety analysis.

- Code generation starting from a safe model.

Both topics represent useful and non-trivial opportunities for an interesting continuation of the research work presented in this thesis.

As a concrete example of a safety concern, consider a software engineer who wants to integrate untrusted components into her software, while preventing these components from rendering dialog boxes on the screen that look too much like a system dialog, because that may persuade the user to reveal his system password. She decides to provide to the untrusted software only a limited set of purpose-designed screen rendering routines, that are relied upon to render everything with a recognizable border decoration and color suite.[1]

Using a safe model for her runtime environment, she could then express approximations for all interacting components, and perform a safety analysis in the model, to either confirm the safety of the design (no native system routines can be abused by the components), or suggest additional restrictions for the integration of the untrusted software. We present in this thesis a method and a declarative language to make this modeling task feasible and practical, together with a prototype implementation of a safety analysis tool.

1.1.1 Motivation

Current operating systems and runtime environments can prevent that a user's data become accessible to another user, but most of them cannot *precisely restrict* a component's authority on a need-to-use basis. The sand-boxing approach in Java (to give untrusted code only limited access to system routines) is but a very rough and static way to impose such restrictions. Software engineers need more refined and dynamic ways to restrict the worst-case influence a software component can have.

Secure software is designed following the principle of least permissions: *"Never give a software component more permissions than necessary to do its job"*. However, because the authority of a component is not only influenced by its own permissions but also by the permissions and the behavior of other components, this design principle should be restated as the *principle of least authority* (Section 1.3.4): *"Distribute permissions among all components and restrict the (programmed) behavior of the relied-upon components, so that no unnecessary authority becomes reachable."*.

Formal safety analysis can be applied in software design to help meet these principles. As a result, the part of a user's software that his safety requirements rely upon (are vulnerable to) could shrink dramatically. In fact, the user should only need to rely on a small security kernel of his operating system.

For instance, a user should not have to rely on his solitaire game to not delete all his files, access his address book, or use his mail system. Sadly, that is still the case for most computer users, because the programs they start, run unconditionally with the user's complete authority, dangerously ignoring the aforementioned principles.

In this short introduction we could not avoid using terminology that was not yet properly introduced. We will clarify these terms in section 1.3, before arriving at the thesis statement.

1.2 Structure of the Thesis

The rest of this chapter consists of an explanation of the concepts that are necessary for a correct interpretation of the thesis statement (Section 1.3), followed by the thesis statement (Section 1.4), and a list of contributions (Section 1.5).

[1]This example is inspired by Stiegler and Miller's: "A Capability Based Client: The DarpaBrowser" [SM02].

Chapter 2 provides an intuitive **overview of the contributions** that are made in this thesis. It is recommended to all readers, regardless of the level of detail at which they want to understand the contents and the relevance of the contributions. The most important concepts used in the following chapters will be introduced. The reader will get a general idea about the context in which the contributions can be appreciated and applied.

Part I contains two chapters that describe and discuss the **foundations** on which this work is built.

Chapter 3 reviews two existing and well known **formal protection models**, that form the basis of the new formalism called "Knowledge Behavior Models". The chapter ends with an overview of the aspects of the relation between permissions and authority, that are important when modeling the propagation of authority in protection systems.

Chapter 4 gives an overview of the main concepts and principles of **capability systems**, and explains how safety properties are enforced by protection systems that depend on holder-managed capability propagation.

Part II contains three chapters that describe the **main contributions** in depth, and a fourth chapter that presents a collection of actual **patterns of collaboration**.

Chapter 5 presents **Knowledge Behavior Models** (KBMs): a new and practical formal approach to calculate safety properties, using safely approximating models and providing the expressive power that is appropriate for engineers of secure software.

Chapter 6 describes the declarative language **SCOLL**: *Safe Collaboration Language*. SCOLL is a language to express patterns of interacting entities and the problems we want to solve considering such patterns. It is based on a kernel language, with a logical semantics expressed in terms of Knowledge Behavior Models (Chapter 5).

Chapter 7 describes **SCOLLAR**: a tool for safety analysis, based on constraint programming, that implements the SCOLL language.

Chapter 8 presents a set of **patterns of interaction and collaboration**, expressed in SCOLL and analyzed in SCOLLAR. It demonstrates the practical applicability of the approach presented in this thesis, not only to analyze patterns in capability systems but also to express, investigate, and compare alternative approaches for building secure software.

Part III contains related contributions in collaboration with other researchers, related work by other researchers, opportunities for future research and conclusions,

Chapter 9 proposes a way to directly express elaborate safety policies as constraints on **authority flow graphs**, derived from an access graph. This chapter is the result of joint work with the inventor of the DomReachability [QVDC06] constraint propagator: Luis Quesada.

Chapter 10 explains the most important **design principles for capability secure multi-paradigm programming languages**, to enable and facilitate the practice of secure programming.

Chapter 11 situates this work in the broader **research context** of related work. It also provides a **research agenda** for future work, listing the opportunities we think are the most important and appealing to apply and extend our work in a broader context.

Chapter 12 presents the **conclusions** to this work.

1.3 Concepts and Definitions

In this section we introduce the concepts necessary for a correct interpretation of the thesis statement.

1.3.1 Protection Systems

We assume that a rule based mechanism is present, that checks and manages *permissions* during the execution of a program. Such a mechanism is called a *protection system* [Bis04]. Permissions indicate what entities can use other entities in what way, during the execution of a program.

The *state* of a protection system, the set of permissions that are valid at a certain stage in the execution of a program, is called the *protection state*. The protection state can change, for instance when entities can grant (delegate, copy) their permissions to other entities. Protection systems allow increasingly more, when increasingly more permissions are given.

Monotonic protection systems [HR78] are protection systems in which the protection state can only grow : permissions are never removed. This means that permissions cannot be revoked.

In the literature, and in most of the examples in this thesis, permissions are *binary* predicates. That means that the protection state can be modeled as a two-dimensional matrix in which the rows indicate the *subjects* (entities that can have permissions), the columns indicate the *objects* (entities to whom the permissions apply), and the cells contain *rights*, indicating the ways in which the subject can use the object. For instance, the predicate `write(alice,bob)` would indicate that the entity `alice` has permission to `write` to entity `bob`, and would be presented in the protection matrix by a `write`-right in the cell at row `alice` and column `bob`.

In this thesis we generalize the concept of permissions, to predicates of *arbitrary* (finite) arity. The reason is not because the generalization would be necessary to model many actual protection systems, but because the restriction to binary permissions is useless as such, and would prevent us from modeling and experimenting with alternative models. For instance, a unary predicate `read(alice)` could be used to concisely and safely approximate a binary predicate `read(alice,_)`, in which the underbar stands for all possible objects.

We will also generalize the concept of protection state, to include everything we consider relevant for safety, not only the permissions. Our generalized protection state may for instance contain aspects of the history of the events and the actions that have occurred, if these aspects are relevant for safety. s

1.3.2 Access Control Lists versus Capabilities

Reference monitors form a crucial part in the implementation of many permission based protection systems. To control the *direct* use of a resource at runtime, a reference monitor will first *identify* the entity that tries to use a resource as the “subject”, then it will find out if that subject has the relevant permission, and consequently it will enable or prevent the use of the resource by the subject.

Typically, the permissions checked by a reference monitor will be organized as “access control lists” (ACLs): lists that correspond to a single resource and enumerate the subjects together with their permissions on that resource. For instance, an ACL for a particular file will state which users have *read* or *write* permission to that file. The

ACL approach is often used at operating system level, where the subject is identified as the human user (principal, group, role) on whose behalf a program or a process is running.

Software engineers need to guarantee the safety in a more refined and dynamic environment, where the subjects no longer correspond to users, but to software entities at the finest level (objects, procedures) that use each other in a certain way, and where all these entities need permission(s) in order to do so. New entities and their permissions are constantly created at runtime. In such conditions, a static ACL based approach is often not practical: the identification of the subjects becomes problematic, and the set of subjects is too dynamic for the approach.

For such refined and dynamic environments, a *capability* based approach is more appropriate. A capability is an *unforgeable reference to a runtime entity, that encapsulates a permission to use the designated entity in a certain way*. Capability systems will be discussed in depth in chapter 4.

If all references to entities are made unforgeable, and all permissions are encapsulated as capabilities, the reference monitor only has to make sure that the rules for acquiring capabilities are respected. Since all the necessary permissions that have to be checked will be encapsulated with the designations that are involved in the acquisition (see chapter 4), and since the designations are necessary in the process of acquisition anyway, the capability approach requires only the overhead to make the references unforgeable, and is completely scalable.

In memory-safe language runtimes (e.g. the Java Virtual Machine), all references are already unforgeable. Therefore, models based on capabilities are a software engineer's natural choice, when analyzing safety in programs written in a language that guarantees such unforgeable references at runtime. *Object capabilities* (Section 4.3) are particularly interesting in this respect, since they have only one type of permission: all the capabilities encapsulate the permission to *invoke* the entity that is designated by the capability.

Besides scalability and ease of implementation, the most important properties of capabilities are:

Discriminative, purposeful use of permissions : With capabilities, entities can use some of their permissions explicitly for a certain task, and others permissions for other tasks. The importance of purposeful use of permissions will be illustrated in section 1.3.5.

Reified permissions : Capabilities encapsulate permissions as unforgeable values. Entities propagate permissions in the same way as other values, e.g. as input and output arguments in invocations.

Combination of Permission with Designation : Capabilities are not just permissions as unforgeable values: they also function as unforgeable designations to the object (target) of the permission. That will dramatically simplify the task of an entity to use the right permission for the right task: the "right" permission *not* necessarily being the permission that allows the task, but the permission that is *expected* to allow the task. Section 1.3.5 will illustrate this advantage.

Remark In the literature, the difference between ACLs and capabilities is sometimes reduced to different ways of organizing the rights in an access matrix: by object for ACLs, and by subject for capabilities. In "Capability Myths Demolished" [MYS03], Miller, Yee and Shapiro show that this distinction is barely relevant and certainly not crucial. It is an over-simplification that can lead to false conjectures about the relative advantages and disadvantages of both approaches.

1.3.3 Permission versus Authority

Permission A permission allows an entity to take an action, like *manipulate-the-screen*, or *get-user-input*, or use another protected resource in a direct way.

Notice that in capability based protection systems, because every permission is combined with a designation, the holder of the capability is not only allowed to perform the action, but is also able to do so. That is not necessarily the case for general protection systems, where `write(alice,bob)` only means: `alice` is allowed to `write` information to `bob`, but she may still have to locate `bob` first.

Authority In [MS03], Miller and Shapiro describe authority as the general *effect* an entity can have in the system. The effect of a permitted action is a simple and direct form of authority, for instance: the effect of the action “manipulate the screen”, when permitted, is: changing the information that is visible on the screen.

But there are other, indirect and potentially complicated forms of authority, that do not correspond to a single action controlled by a permission. For instance, in figure 1.1, entity `alice` can and will `read` instructions from `bob`, and interpret the instructions to `manipulate` the `screen` accordingly. The resulting *authority* is the same as if `bob` would have had a permission to `manipulate` the `screen`.

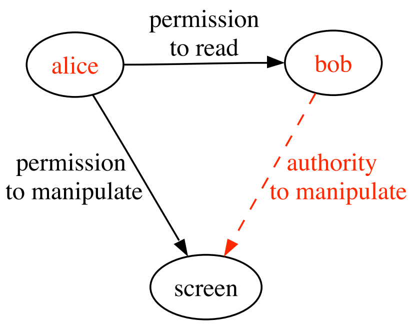


Figure 1.1: Indirect Authority: `bob` can manipulate the screen indirectly.

The relation between permissions and authority is causal: ultimately all authority is brought about by the use of permissions. But that relation has many aspects that are important when modeling protection systems. Section 3.4 discusses these aspects in detail.

Notation in Authority Graphs

In illustrations like figure 1.1 we will indicate the names of relied upon subjects in black and the names of untrusted (or unknown) subjects in red. Solid black arrows indicate permissions, while dashed red arrows indicate (indirect) authority, and grey arrows will indicate invocations (e.g. figure 3.4).

1.3.4 The Principle of Least Authority (POLA)

The principle of least authority states that every entity should have no more authority than is necessary for the functionality that it is supposed to provide. It is an extreme interpretation of Saltzer and Schroeders "Principle of Least Privilege" [SS73], that is often interpreted in a weaker form as the "Principle of Least Permissions". In view of the many and complex ways in which authority can be brought about, POLA is to be used as a guideline when developing secure software, rather that as a strict rule.

For instance, if a software component `alice` needs to manipulate a particular area of the screen, it may be in accordance with the principle of least permissions to give `alice` a permission to manipulate the screen, but it would not be strictly in accordance with the principle of least authority. In an (object) capability system, we can do better, and give `alice` a capability that designates a relied-upon programmed entity `screen proxy`, who can be *relied upon* to relay the manipulation instructions to the screen, only after cropping them to the restricted area. Figure 1.2 illustrates the situation.

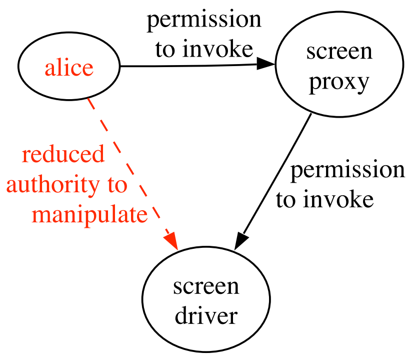


Figure 1.2: Indirect Authority: `screen proxy` is relied upon to reduce `alice`'s authority to manipulate the screen.

Capability based safety enforcement will rely very strongly on programmed entities, to accurately reduce the authority that they make available to the other entities they interact with.

As a direct consequence of POLA, all authority that is available to an entity "ambiently" (without requiring an explicit transfer of authority by another entity), is to be avoided. Ambient authority is prevented in language runtimes like the *E* virtual machine [MSC$^+$01], where the loader makes sure that all software is loaded with no access to any other runtime entity that can provide authority.

1.3.5 Safety Enforcement with Protection Systems

To be effective, safety enforcement should prevent all illegal authority, regardless of the ways in which such authority can be brought about (what permissions are involved). Three mechanisms are involved, when restricting authority:

1. Enforcing the permissions: make sure that no illegal actions can be performed. This is a task for a reference monitor at runtime, and as we have seen already (Section 1.3.2), that task is simple for capability based protection mechanisms.

2. Restricting the propagation of permissions: make sure that permissions can only propagate in legal ways. This task is performed at runtime by the reference monitor too. “Propagate a permission” can be a permission itself. Often, the reference monitor will have to check more than one permission. For instance, the reference monitor may have to enforce a rule that says: *an entity can only grant the permissions it has itself.* This is the well known principle of attenuation.

   Object capabilities have again a simple and valid solution to this problem, based on the fact that “invoke” is the only permission and on the inextricable combination of permission and designation into a capability. A memory safe language runtime will suffice to do the job (Section 4.3)

3. Allow discriminative, intentional application of permissions. The protection system must allow the entities to choose the permission(s) they want to use to perform a certain task. Otherwise, the entities cannot protect themselves and their clients from their other clients’ bad intentions or programming errors.

   For instance, suppose that `alice` and `bob` make use of a computation service `carol` as illustrated in figure 1.3. The results will be written by `carol` to the file that is designated by her client. In the process, `carol` will also consult and update her own file. When writing output for `alice`’s calculations, there must be a way for `carol` to indicate to the protection system that only `alice`’s permissions should be used. Otherwise, `alice` could choose `bob`’s file or `carol`’s file as output, and disturb `carol`’s and `bob`’s correct workings.

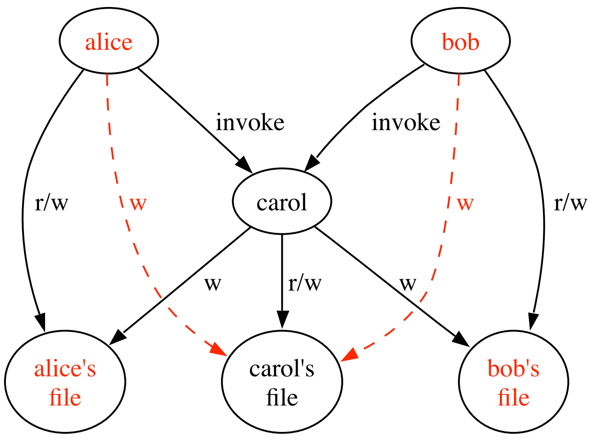


Figure 1.3: A service entity (`carol`) than cannot be abused.

The solid arrows indicate permissions (w for “write” and r for “read”). The dashed arrows indicate the *reduced* $write$-authority `bob` and `alice` have on `carol`’s file: they can influence the content of the file, but only by using `carol`.

We can take two approaches to prevent illegal use of `carol`:

(a) Indirect use and delegation of permissions:
Only the protection system can manipulate the permissions as values. It prevents illegal use of `carol` by revoking and reactivating her permissions when needed.

(b) Direct use and delegation of reified permissions:
Permissions become unforgeable values, for the entities to use and delegate explicitly and on purpose. Now `carol` can prevent her clients from using her in an illegal way, by demanding them to provide such permissions and by using these delegated permissions for the right purpose.

Approach (a) is difficult to implement. The protection system, possibly with the help of `carol`, must find out on whose behalf `carol` is using her permissions.

For instance, when `alice` invokes `carol`'s service, `carol` could tell the protection system to disable all `carol`'s permissions, except for the ones she shares with `alice`, before writing to the file that `alice` chose. For `carol` to be able to log the action to `carol's file`, `carol`'s permissions would need to be re-activated again.

The protection system can apply dynamic delegation (delegation of `alice`'s permissions to `carol`, but only in the context of `alice`'s invocation of `carol`), and stack walking (inspect the invocation stack to find out who is invoking whom) [WBDF97], but finding out exactly on whose behavior `carol` is using her permissions will still be hard. Practical solutions are approximative at best (see sections 8.1.2 and 8.5.4).

Approach (b) is already implemented by capability based protection systems. Capabilities function at the same time as designation to the target of a permission and as that permission itself, reified as an unforgeable value entities can use and convey purposefully.

`carol` can require her clients to communicate their choice of output file as an argument to the service request, in the form of a *write*-capability to that file. If `alice` chooses a file she has no write permissions to, `carol`'s attempt to write output to that file will fail when she tries to use the value provided by `alice` as a *write*-capability.

To enable `alice` to comply with `carol`'s requirement, capabilities allow entities to delegate their permissions as values. It suffices for `alice` to have two capabilities: one to *invoke* `carol`, and one to *write to* `alice's file`, to be able to delegate her file-capability to `carol`.

An important advantage of permissions as values is: the software developer can use the normal invocation and scoping mechanisms to regulate delegation and revocation. `carol` accepts capabilities as input arguments to a computation request, and can drop them afterwards, simply by letting the capabilities go out of scope. No other mechanism for dynamic delegation is necessary.

The main advantage of relinquishing delegation power to the entities, is that POLA can be applied very strictly: `carol` will not need permissions to all her client's files. The clients themselves are expected to convey the necessary capabilities to `carol`.

1.3.6 Behavior based Safety Analysis

Because of the many complex ways in which authority can be brought about, an up-front worst-case analysis of the safety requirements is necessary. This thesis proposes a practical approach to safety analysis, based on an explicit model of the software that models the following aspects of safety enforcement:

1. How the protection system restricts the propagation of permissions.
2. In what circumstances the relied-upon entities (do not) use their permissions.

The second aspect is referred to as *behavior*. In "Robust Composition" [Mil06b] Miller gives an overview of approaches to safety analysis, and describes this approach as: calculating a *behavior-based bound on eventual authority*. Miller argues that this approach is preferred, because it allows a tractable calculation of a non-trivial upper bound on the authority that can be reached, that is "interesting" because it gets more accurate when the behavior is modeled in greater detail.

1.3.7 Safe Approximation of Behavior

In this work, we use models that safely approximate the authority propagation in the software to be implemented (or analyzed) by modeling a monotonic approximation to the behavior of the software entities. This means that we will not be able to express directly in the model that a software component *no longer* uses a permission in conditions it did use the permission before (the behavior based equivalent of revocation). This is a restriction in our approach, that is not absolutely necessary to make the safety analysis tractable. Therefore, non-monotonic refinements are considered to be interesting future work.

However, the monotonic approach has the practical advantage that safe approximations of the behavior of a software entity at runtime can easily be inferred from program code.

Chapters 2 and 5 will give an intuition about how the process of safely approximating behavior goes about. Several detailed examples will be provided in the chapters of part II. The process is designed to allow incremental refinement of the model, starting with a crude but safe description of the behavior of the relied-upon entities, and refining that model step by step, if and where the analysis indicates that more refined behavior is necessary to guarantee the safety requirements.

1.3.8 Designing for Safety, Relying on Collaboration

In object capability systems, no single entity can bring about authority on its own. If `alice` has no capabilities, she has to wait for another entity to invoke her. But even if `alice` has a capability designating `bob`, she can invoke `bob`, but the effect of that invocation will depend on `bob`.

For instance, if `alice` has access to `bob` and `carol`, she may want to introduce `carol` to `bob`, by invoking `bob` with the argument `carol`. `Bob`'s behavior may be completely non-collaborative and effectively annihilate the effect of the invocation, either by ignoring the invocation completely, or by accepting access to `carol` but never ever using or propagating the accepted capability.

The term "*collaboration*" in the title of this thesis will be used to refer to protection systems in which no authority can be brought about by an entity without the collaboration of another entity. Collaborative protection systems make its easier to design software in which the safety requirements can be enforced by relied-upon entities, strategically inter-positioned between the untrusted entities our software product must rely upon for its intended functionality, but not for ensuring its safety.

In non-collaborative systems, two untrusted entities that have access to a third entity cannot be restricted in their authority by the third entity. For instance, if `alice` and `bob` are untrusted entities that have read and write permissions to `carol`, and `carol`'s collaboration would not be needed to turn these permissions into actual authority, `carol` would not be able to prevent `alice` and `bob` from communicating with each other by using `carol` as a channel. If `carol`'s collaboration would be necessary she could fine tune the authority in the way she wanted, for instance by giving her clients read authority only to data that was not written to her by her clients.

Secure programming is not easy, but collaborative systems at least provide a powerful mechanism to ensure POLA: by programming relied-upon components to make sure that no excess authority leaks to the untrusted components that have access to them.

This thesis contains many examples of successful applications of this approach. Some examples will simply prove that the relied-upon entities effectively guarantee the safety requirements. Other examples will investigate possible alternatives for relied-upon behavior, and provide insight in the conditions in which a pattern can be safely applied in our software. Occasionally, we go even further and *generate* abstract *patterns of collaborating entities*, to guarantee certain safety requirements. The abstract patterns in this thesis correspond to Miller's concept of "programmable abstractions for access control" [MS03].

This work can be situated between software verification and the design of security protocols. In software verification, abstract interpretation is applied directly to the program code, to prove or refute general software requirements. In design and verification of security protocols, abstract models of lower complexity are tested for safety. In this work, abstract models of adaptable complexity are used to prove safety, and to analyze the influence of relied-upon behavior on provable safety.

1.4 Thesis statement

Patterns of collaborating entities provide a practical way to design provably secure software, in analogy to the use of design patterns for developing maintainable software. A behavior based analysis of eventual authority in these safety patterns allows the software engineer to reason about the necessity and effectiveness of protective measures, such as method interception and the interposition of software components that are relied upon to use their access permissions in restricted ways. In capability-based runtime environments that provide no ambient authority, the strategic interposition of relied-upon components is sufficiently restrictive to enforce many practical safety strategies, without the need for an ACL-based reference monitor.

1.5 Contributions

This section lists the contribution of this thesis. Chapter 2 will give an extended introduction to the main contributions.

1.5.1 Major Contributions

1. The introduction of a new and practical formalism, *Knowledge Behavior Models*, that allows reasoning about safety requirements in patterns of interacting entities, and has the required expressive power to be useful in secure software engineering.

2. The flexible technique of modeling by *aggregation*, whereby (potentially infinitely) many runtime entities are modeled into a single subject whose behavior is the union (lowest upper bound) of the individual behaviors. Aggregation makes the model simple and tractable. We prove that this modeling technique results in a *safe approximation*.

3. The domain specific declarative language SCOLL in which abstract patterns of interacting entities and their safety requirements can be expressed, with a logical semantics in terms of Knowledge Behavior Models. An iterative, incremental approach to behavior model refinement, starting with a safe but possibly too crude model, and gradually refining the model (while keeping it safe) when and where necessary.

4. The development of a prototype safety analysis tool *SCOLLAR*, based on constraint programming, that implements the language SCOLL, including a web-based version of the tool.

5. The application of SCOLL and SCOLLAR to several real world problems concerning the safety and applicability of design patterns for secure software, in capability based and ACL based paradigms, as a successful test of the thesis statement.

6. A general contribution to the understanding of the nature and the properties of capabilities, their formal representations, and the possibilities for applying capability based security.

1.5.2 Minor Contributions and Side Contributions

1. An analysis of the possible relations between authority and permissions in protection systems.

2. An exploration of authority flow graphs, as a possible complement to the rule-based approach in the main contribution.

3. A motivated list of design principles for the design of capability-based, secure, multi-paradigm languages.

Chapter 2

Overview of the Contributions

This chapter gives an intuitive overview of the main contributions in this thesis. This chapter is recommended to all readers, regardless of the level of detail at which they want to understand the contents and the relevance of the contributions.

The most important concepts used in the following chapters will be introduced, even if some of them may need revisiting in later chapters. The reader will get a general idea about the context in which the contributions can be appreciated and/or applied. No proofs are included, as the contributions will all be discussed in depth in their own chapter.

The examples in this chapter are simple and introductory. Practical and elaborated examples are developed and discussed in chapter 8.

2.1 A New Formal Model for Safety Analysis

In safety analysis we investigate the relation between sets of possible actions and their potential effects to understand what actions should be allowed or disallowed, in order to avoid the effects that are unwanted or illegal. As in many other fields, the relation between causes and effects can be very complex and dynamic, for instance because the effects of possible actions can make more actions possible. We will call the potential effects of the actions: "authority".

Even when the actions are controlled by permissions, guarded and enforced by a dedicated mechanism (a protection system), there is no guarantee that this relation is even computable. This was shown in 1976 by Harrison, Ruzzo, and Ullman [HRU76].

This is an important result, but it should not discourage us to search for tractable models that can approximate the action-effect relation from the safe side: possibly over-estimating the potential effects but never under-estimating them. The tractable models we encountered in the security literature are not practically useful for software engineers to model the problems that are of concern to the safety in their software.

The Knowledge Behavior Systems (KBMs) proposed in this thesis represent a new approach to model tractable and safe approximations, and were designed from the start with these goals in mind:

1. Model not only the potential effects on the distribution of permissions, but all potential effects that are relevant for safety, including indirect potential effects like the indirect authority in figure 1.1.

2. Model not only the restricting effects of permissions, but also the restricted use of the permissions by the relied-upon parts in the software.

3. Provide flexible expressive power to model different parts in the software at their own most appropriate level of detail.

4. Support the incremental refinements of existing models.

2.1.1 An Extended View on Protection States

In "Computer Security: Art and Science" [Bis04] Matt Bishop defines the *protection state* as that part of the state of the computer system that is relevant for protection. Bishop defines the *state* of a computer system as "the collection of the current values of all memory locations, all secondary storage, and all registers and other components of the system".

This definition is too restrictive for our purposes, as it may not include all situations that are relevant for safety, for instance if safety not only depends on the current state but also on the previous states. For instance, it may be dangerous for the system to traverse a certain series of states, because that could correspond to a message being sent to an output device which could trigger a nuclear war.

We extend the term protection state to refer to all aspects that are relevant for security, regardless of whether they can be derived from the current state of the computer. Our models will of course be state transition systems, but their states will not necessarily correspond to predicates about the current state of a computer system.

We also want to include not only the permissions in the protection state, but also other forms of authority. Effects achieved by the orchestrated use of several permissions by different entities may be as dangerous as the direct use of a single permission.

It was Bishop himself who, together with Snyder, identified the importance of indirect authority in their paper on Take-Grant models (Take-Grant systems) in 1977 [BS79]. In that paper, the concept of "de facto right" corresponds to indirect authority. KBMs are an extension of Take-Grant models. The latter will be described in depth in section 3.2.

Extending the Dimensions of the Permission Matrix

Traditionally, the protection state is depicted as a two dimensional matrix in which the rows depict the subjects (the entities that can have rights) and the columns depict the objects (the entities the subject's right can apply to). The restriction to two dimensions is not practical to model all the aspects of the extended protection states in a KBM.

Besides permissions and authority, our protection states must incorporate the restricting influence of the relied-upon entities on the potential effects of possible actions. Some entities may be relied upon to refrain from using some of their permissions, turning possible actions into actions that will never be actually performed. For instance, Take-Grant models have a simple concept of *passive subjects*, called "objects", who never use any permission they have.

In some systems, entities may have the power to diminish or annihilate the potential effects of possible actions. For instance if `alice` has invoke-permission to `bob`, `bob` may be programmed to make no important changes to the protection state, and the safety analysis could rely on that.

To allow software engineers to take both kinds of restrictions into account when designing safe software or when analyzing existing software for safety, KBMs will extend the protection state from a set of binary permission-predicates to a set of predicates of arbitrary arity. Moreover, all entities will be modeled as subjects and objects will be a particular type of subjects that cannot (or will not) accept permissions.

The Protection State in KBMs

We use predicates to present three different aspects of the protection state:

Permission predicates : usually binary, representing the traditional rights matrix. For instance, the predicate `access(A,B)` could be used to express the permission of subject `A` to access subject `B`.

We will not assume that the holder of a permission is directly aware of his permissions. A subject's awareness of its environment will be modeled explicitly with knowledge predicates. Only the protection system is directly aware of permissions between subjects. For all practical purposes, it would not matter if a KBM would express this permission conversely as: `accessibleBy(B,A).`

Behavior predicates : of arbitrary arity, representing the positive influence a subject has on the actual use of a permission and/or on the effect of using a permission. For instance, the predicate `sendTo(A,B,X)` could be used to express that subject `A` is willing to invoke subject `B` with input argument `X`.

To underline the fact that it is subject `A`'s behavior, we will propose an alternative notation that puts the first argument in front of the predicate label, separated by a colon. By convention, we often prefix the label of a behavior predicate with "`may.`" to indicate that we model possible behavior: behavior that we cannot with certainty exclude, due to our limited knowledge of the entity's behavior restrictions, or to our safe (over-)approximation of the entity's behavior in the KBM.

A behavior predicate will usually have this form: `A:may.sendTo(B,X)`

Knowledge predicates : of arbitrary arity, representing potential effects. For instance, the predicate `sentTo(A,B,X)` could be used to model the potential effect on subject `A` of having successfully invoked subject `B` with input argument `X`.

Like behavior predicates, knowledge predicates have an alternative notation, to underline what entity has the knowledge. By convention, we usually prefix the label with " `did.`" to indicate that we model knowledge about a successful interaction that a subject can rely upon with certainty. The uncertainty is only in our approximation of behavior, not in our model of the effects of behavior.

A knowledge predicate will usually have this form: `A:did.sendTo(B,X)`

2.1.2 Refining the Protection State

Once a KBM has been defined, it will be relatively easy to refine the model. The main reason to refine the behavior of a relied upon entity is to express its restricted use of permissions more accurately.

For instance, instead of using the predicate `B:may.return(Y)` to express `B`'s possible willingness to return `Y` when invoked, we may need a more refined predicate to express that `B` only returns `Y` when `B` received `X` in the same invocation. That could be expressed by the refined predicate: `B:may.exchange(X,Y).`

KBMs provide a general framework that allows the software engineer to incrementally add refined behavior and knowledge, without having to re-model the whole KBM.

2.1.3 A Fixed Set of Subjects for a Finite Protection State

For the purpose of proving safety and finding safe abstract patterns, it is crucial that the calculation of the eventual authority (potential effects) in the model is tractable. That is the main reason why KBMs have a fixed, finite set of subjects.

For a KBM to be representative, all software entities must be modeled into this fixed set of subjects. Subjects that model more that one entity are called "aggregated" subjects. For aggregation to be safe, every predicate in the KBM that concerns an aggregated subject must represent the possibility that the predicate holds in the software for at least one of the entities aggregated into the subject. In other words, it must be true for the subject in the model as soon as it is possible for one of the entities in the software.

For instance, subject `bob` could be used to model a certain software entity, together with all the entities it may possibly create at runtime. Then, the predicate `access(bob,alice)` will indicate the possibility that one of the entities aggregated into `bob` has access to one of the entities aggregated into `alice`. Section 5.7.4 will prove that aggregation is a safe approximation approach.

Section 5.3.1 will show why and when aggregation is useful, and explain different aggregation strategies. An elaborated example of a less trivial aggregation strategy is presented in section 8.3.1.

2.1.4 Modeling Protection State Transitions

A protection system is not static: its protection state will change when actions are performed. Rules in the protection system will express what actions are possible and what potential effects every action has on the state of the protection system. In other words, the rules define protection state transitions.

In traditional protection systems, the permissions are the (only) preconditions for the rules, and the potential effects are all modeled as changes to the permission matrix. For instance, the rule

```
grant(A,B) ∧ read(A,X) ⇒ read(B,X)
```

would indicate that, if `A` has *grant*-permission to `B` and *read*-permission to `X`, then the potential effect is that `B` also gets *read*-permission to `X`, because it was granted by `A`.

Compared to such permission-centric models, Take-Grant models take two important steps towards more expressive protection state transitions:

1. The restricted influence of behavior is taken into account. For instance, the Take-Grant rule "grant", that propagates "de jure rights" (permissions), is only valid for active granters:

   ```
   grant(A,B) ∧ read(A,X) ∧ A:active() ⇒ read(B,X)
   ```

2. The potential effects of the rules can present all kinds of authority, not just permissions. For instance the Take-Grant rule "spy", that propagates "de facto" `read`-authority, can be represented as:

   ```
   read(A,B) ∧ read(B,C) ∧ A:active() ∧ B:active()
   ⇒ deFactoRead(A,C)
   ```

These two extensions bring Take-Grant models very close KBMs. Two more extensions will be added:

3. The behavior of the relied upon subjects will no longer be restricted to active or passive, but can be expressed to arbitrary precision. For instance, the following rule could model the propagation of access permissions in an invocation-based interaction model:

   ```
   access(A,B) ∧ access(A,X) ∧ A:may.sendTo(B,X)
   ∧ B:may.receive()
   ⇒ access(B,X) ∧ A:did.sendTo(B,X) ∧ B:did.receive(X)
   ```

4. The behavior of a subject is no longer static, but can be adapted, in response to what the subject can come to know about its environment. For instance, subject `alice` can be programmed to invoke the subjects she has accepted when being invoked. That could be expressed in a KBM as the following rule, applying only to `alice`'s behavior:

   ```
   alice:did.receive(X) ⇒ alice:may.sendTo(X,_)
   ```

In KBMs, the rules that derive potential effects from possible actions are called *system rules*. The rules that express a subject's reaction to its environment are called *behavior rules*. Behavior rules can only model dynamic behavior that monotonically increases when the subject receives more knowledge.

Remark The *de-jure* rules in Take-Grant systems were viewed as a way to analyze dataflow between entities. However, because data can also flow via covert channels [Lam73], such analysis is not safe. The knowledge predicates in KBMs are not used for data flow analysis. They do not represent knowledge of an arbitrary fact, but knowledge of a fact of which the subject also knows that it was received via a legal interaction. Absence of such knowledge will never keep untrusted subjects in the KBM from deploying their full behavior.

2.1.5 Knowledge Behavior Models

Definition 1. ***KBM***
A KBM is a tuple $\langle S, P_p, P_k, P_b, Sys, Beh\rangle$, in which:

- *S is a finite set of subjects.*
- *P_p is a set of permission predicate symbols with their arity*
- *P_k is a set of knowledge predicate symbols with their arity*
- *P_b is a set of behavior predicate symbols with their arity*
- *Sys is a set of system rules*
- *Beh is a set of behavior rules*

Because no negation is allowed in the rules of a KBM, the logical semantics are simple and equivalent to datalog [GM78].

2.1.6 Safety Analysis with KBMs

KBMs specify a logical derivation, from an initial configuration as a finite set of predicates, to the complete set of all predicates that are reachable in the model. If a KBM models a safe approximation of the permissions, the behavior, and the authority propagation in a software program, the authority that cannot be derived in the model will correspond to authority that cannot be reached in the software.

Therefore KBMs allow us to express and compute safety problems in the software. To express safety problems, we define three more concepts:

Definition 2. ***Configuration***
A configuration in a KBM is a set of predicate facts (grounded predicates), using predicates that are defined in the KBM, over subjects that are defined in the KBM.

Definition 3. ***Safety properties***
The safety properties of a KBM are the predicates that cannot be derived from a given configuration by the rules in the KBM.

Definition 4. ***Liveness possibilities***
The liveness possibilities of a KBM are the predicates that can be derived from that configuration by the rules in the KBM. Contrary to safety properties, liveness possibilities provide no proof that the authority will be reached or can be reached in the modeled software. They will only be used to restrict the search for safe subject behavior and safe configurations to those solutions in which the restrictions do not exclude *the liveness possibilities.*

The following types of safety problems will be considered and solved in this thesis:

Simple Safety Problem : Given a KBM, an initial configuration for the KBM, and a set of safety properties, the problem is: "Do the safety properties hold?"

In other words, is it impossible to derive *any* of the safety properties from the KBM and the initial configuration.

Practical Safety Problem : Given a KBM, an initial configuration for the KBM, a set of safety properties, and a set of liveness possibilities, the problem is: "Do the safety properties and liveness possibilities hold?"

In other words, is it possible to derive *all* the liveness possibilities and at the same time impossible to derive *any* of the safety properties from the KBM and the initial configuration?

Behavior Maximization Problem : Given a KBM, an initial configuration for the KBM, a set of safety properties and liveness possibilities, and a set of subjects whose behavior must be maximized, the problem is: "What are the possibilities to maximize the behavior of the given subjects, given that the safety properties and liveness possibilities should hold?"

A behavior maximization problem asks for solutions that maximize the cooperative behavior of a set of relied-upon subjects. It can have many solutions.

Knowledge Maximization Problem : Given a KBM, a minimal initial configuration for the KBM *min*, a maximal initial configuration for the KBM *max*, and a set of safety properties and liveness possibilities, the problem is: "What are the possibilities to maximize the initial configuration between *min* and *max*, given that the safety properties and liveness possibilities should hold?"

A knowledge maximization problem asks for solutions that maximize the initial configuration. It can have many solutions.

Generalized Maximization Problem : Given a KBM, a minimal initial configuration for the KBM *min*, a maximal initial configuration for the KBM *max*, a set of subjects whose behavior must be maximized, and a set of safety properties and liveness possibilities, the problem is: "What are the possibilities to maximize the initial configuration between *min* and *max* and the behavior of the given subjects, given that the safety properties and liveness possibilities should hold?"

A generalized maximization problem combines a behavior maximization problem with a knowledge maximization problem. It can have many solutions.

2.2 Safe Collaboration Language: SCOLL

This section provides an intuition about the declarative language SCOLL. SCOLL will be used to describe abstract patterns of interacting subjects, and to express the safety problems we want to solve considering such patterns.

Chapter 6 presents SCOLL as an extensible language, based on a kernel language with a logical semantics that is expressed in terms of KBMs.

2.2.1 Example: The Caretaker

To illustrate the purpose of SCOLL, we will program a simple version of the caretaker pattern and explain the intuition behind the SCOLL program. This pattern is derived from Redell's [Red74] pattern of the same name. An extended version of the caretaker pattern will be examined in section 8.2.

The initial situation in the pattern is depicted in figure 2.1.

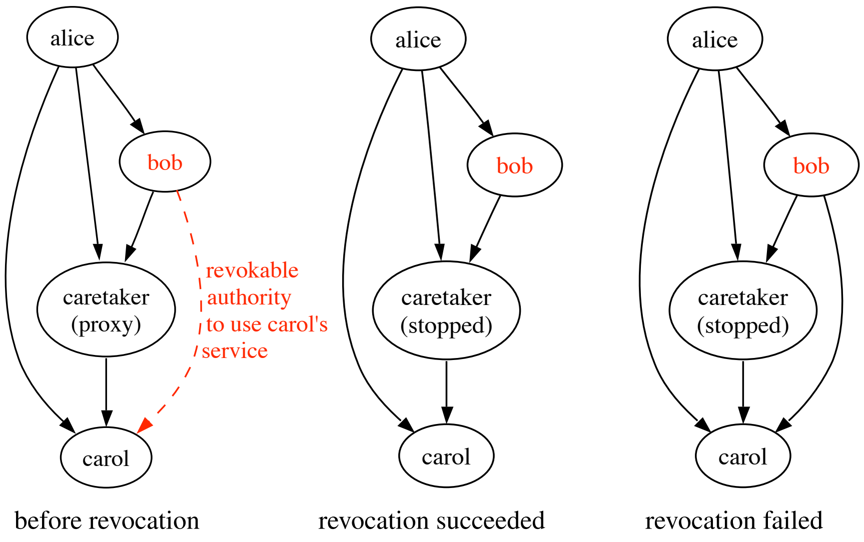


Figure 2.1: The caretaker pattern, before and after revocation.

The left part of figure 2.1 shows the initial situation. A subject `alice`, with access to `bob` and to `carol`, wanted to give `bob` *revocable* authority to `carol`'s services.

Instead of introducing `carol` to `bob`, she has introduced `bob` to a proxy subject, `caretaker`. The `caretaker` is relied upon to:

- act as a transparent forwarder: forward all requests to `carol`, and return what `carol` returns to the request.
- stop acting as a transparent forwarder as soon as the *revoke* instruction is given, in order to end `bob`'s authority to use `carol`'s services
- do nothing else

The middle part of figure 2.1 shows the situation after `alice` has told `caretaker` to stop proxying: `bob` no longer has the authority to use `carol`'s service, because the `caretaker` stopped collaborating. The rightmost part of figure 2.1 shows the situation to be avoided: the revocation fails here since bob no longer needs `t`he `caretaker` because he has got direct access to `carol` and can invoke `carol`'s services directly.

For this simplified example, we will assume that `alice` will not use her permissions after the initial set-up, except for eventually giving `caretaker` the `revoke` instruction.

These are the safety problems we want to solve:

1. Can we prove that, regardless of `carol`'s behavior, `bob` can never get direct access to `carol` (right part of figure 2.1) to render `alice's` revocation efforts futile?
2. If we cannot prove that, we want to know on what restrictions in `carol`'s behavior we have to rely, to guarantee that `bob`'s authority will remain revocable.

The complete caretaker pattern analysis in section 8.2 will also take `alice`'s potential influence into account.

2.2.2 Code Example for the Caretaker Pattern

Let us consider two code examples that correspond to the caretaker pattern. The code examples will help us to understand the SCOLL program that we will use to describe the caretaker pattern.

Figure 2.2 shows the pattern in Oz-E [SV05], the capability-secure variant of Oz. A capability secure language is a memory safe language, in which no entity (object, procedure, closure, ...) can have access to an entity that provides authority, without proper introduction by an other entity. In other words, entities have no authority when they are loaded. Oz-E is future work, and its major design goals are presented in chapter 10.

Note that, to be compatible, we made all entities functions with one input and one output argument. `Alice` does not use its arguments, but only invokes `Bob` with the input argument `Caretaker`. That will allow `Bob` to use the caretaker.

`Bob`'s code is unknown, and we will make no assumption about it. Therefore we must assume that Bob will do everything to break our safety requirements.

`Caretaker` is a classical proxy, apart from the fact that it checks its private boolean variable `Proxying`, before forwarding any invocation. `Proxying` will become false as soon as someone sends `revoke` to the caretaker.

```
Alice = proc{$ In Out}
           {Bob Caretaker _} % introduce caretaker to bob
           {Delay 60000}
           {Caretaker revoke _}
        end
Bob = {{Link ["bobModule.ozf"]}.1.makeBob}
Caretaker = local Proxying = {NewCell true}
            in
               proc{$ In Out}
                  if In == revoke then Proxying := false
                  elseif @Proxying then {Carol In Out}
                  end
               end
            end
Carol =  proc{$ In Out}
           . . .
         end
```

Figure 2.2: Simple caretaker in Oz-E.

2.2.3 SCOLL code for the Caretaker Pattern

Figure 2.3 shows the pattern (program) in SCOLL. SCOLL programs always exist of exactly six parts, identified by the following keywords:

1. `declare` : the predicates that will be used in the program
2. `system` : the system rules that govern authority propagation
3. `behavior` : the different *types* of behavior that will be used to model the behavior of the subjects
4. `subject` : the subjects that play a role in the pattern
5. `config` : the initial configuration of the pattern
6. `goal`: the safety properties and the liveness possibilities

Before we can give an intuition about the meaning of every part in the SCOLL program, we have to consider the interaction model: how the subjects in the pattern interact and how this interaction can influence the propagation of access.

2.2.4 The Interaction model

By themselves, the graphs in figure 2.1 give us no indication of how access can propagate in the pattern. However, since our analysis is going to be applied to software in Oz-E, we have all we need to model a safe approximation of the interaction model.

It is important that we model all possible ways in which access can propagate in these languages, but we don't have to do that into detail. Moreover, since Oz-E is a capability secure language, we know that no access can be gained without interaction (See chapter 10).

All interaction mechanisms in these languages – whether they involve actual invocation of functions or procedures, message sends between objects, assignments or

```
declare
   permission:  access/2
   behavior:  may.sendTo/3 may.getFrom/2 may.return/2
              may.receive/1
   knowledge:  did.sendTo/3 did.getFrom/3 did.return/2
               did.receive/2
system
   access(A,B) access(A,X) B:may.receive()
   A:may.sendTo(B,X)
   => access(B,X) A:did.sendTo(B,X) B:did.receive(X)
   access(A,B) access(B,X) A:may.getFrom(B)
   B:may.return(X)
   => access(A,X) A:did.getFrom(B,X) B:did.return(X);
behavior
   UNKNOWN: { => may.sendTo(A,X) may.getFrom(A)
                 may.return(X) may.receive();}
   MINIMAL: {}
   ALICE: {isBob(B) isCaretaker(C) => may.sendTo(B,C);}
   PROXY: { => may.receive();
          isCarol(C) => may.getFrom(C);
          isCarol(C) did.receive(X) => may.sendTo(C,X);
          isCarol(C) did.getFrom(C,X) => may.return(X);
subject
   alice:  ALICE
   bob:  UNKNOWN
   caretaker:  PROXY
   ?  carol:  MINIMAL
config
   access(alice,alice) access(alice,bob)
   access(alice,caretaker) access(alice,carol)
   access(bob,bob)
   access(caretaker,caretaker)
   access(caretaker,carol)
   access(carol,carol)
   alice:isBob(bob) alice:isCaretaker(caretaker)
   caretaker:isCarol(carol)
goal
   !access(bob,carol)
```

Figure 2.3: Simple caretaker in SCOLL.

bindings – can be modeled abstractly as invoke-respond interactions where an invoker takes the initiative to interact with a responder. The propagation of access during an abstract invocation can go in either direction: from invoker to responder or from responder to invoker.

A simple model for abstract interaction is this:

1. If and entity `A` has access to a entities `B` and `X`, and `A` initiates an interaction with `B` to pass `X` to `B` and `B` accepts access when it is invoked, then the propagation succeeds. As a result, `B` will also have access to `X`, and `A` and `B` will both have knowledge about successful interaction.

2. If and entity `A` has access to an entity `B`, and `B` has access to an entity `Y`, and `A` initiates an interaction with `B` and `B` return access to `Y` when it is invoked and `A` accepts access from its responders, then the propagation succeeds. As a result, `A` will also have access to `Y`, and `A` and `B` will both have knowledge about successful interaction.

With this interaction model in mind, we can now explain the two first parts of the SCOLL program in figure 2.3.

2.2.5 The `declare` Part.

The predicates defined in this part will form the basic parts of all the SCOLL statements. All predicates are declared in the form: `<label> ``/'' <arity>`, as can be seen in figure 2.3.

Three kinds of predicates are distinguished, and indicated by their proper keywords:

1. `permission` : Only one predicate is declared here: `access/2` with arity 2, to model the only conditions we have in our interaction model that do not depend on the way the entity is programmed. Because Oz-E is capability secure, access is neither ambient nor forgeable and is therefore a real permission.

2. `behavior` : The other conditions we have in our interaction model are behavior predicates and must be declared here:

   - `may.sendTo/3` : a ternary condition derived from the first rule in our interaction model. `A:may.sendTo(B,X)` indicates that subject `A` is willing to invoke `B` and pass `X` as an input argument to the invocation.
   - `may.getFrom/2` : a binary condition derived from the second rule in our interaction model. `A:may.getFrom(B)` indicates that subject `A` is willing to invoke `B` and accept all access that `B` will return from that invocation
   - `may.return/2` : a binary condition derived from the first rule in our interaction model. `B:may.return(X)` indicates that subject `B` is willing to respond (`r`) when being invoked, by passing `X` to the invoker.
   - `may.receive/1` : a unary condition derived from the second rule in our interaction model. `B:may.receive()` indicates that subject `B` is willing is willing to respond when being invoked, by accepting all access the invoker will pass in the invocation.

3. `knowledge` : Here we declare the predicates that will model all possible effects of successful interaction in our interaction model, except for the propagated

permissions. These predicates are called "knowledge" predicates, because they represent knowledge an entity can have about a successful interaction it was involved in.

When modeling the restricted behavior of relied-upon entities, we will use knowledge predicates to specify the conditions in which the entity is willing to cooperate in an interaction.

- `did.sendTo/3` : a ternary predicate derived from the first rule in our interaction model. `A:did.sendTo(B,X)` is knowledge for subject `A`, indicating that it invoked `B` and passed `X` to `B`.
- `did.getFrom/3` : a ternary predicate derived from the second rule in our interaction model. `A:did.getFrom(B,Y)` is knowledge for subject `A`, indicating that it invoked `B` and accepted access to `Y` returned by `B` in that invocation
- `did.return/2` : a binary predicate derived from the second rule in our interaction model. `B:did.return(X)` is knowledge for subject `B`, indicating that it was invoked and responded by passing `X` to the invoker.
- `may.receive/2` : a binary predicate, derived from the first rule in our interaction model. `B:did.receive(X)` is knowledge for subject `B`, indicating that is was invoked and responded by accepting access to `X` from its invoker.

Note that we did not include the invoker in the knowledge available to the responder, because in Oz-E the responder does not get access to the invoker.

2.2.6 The `system` Part

This part must contain a list of *system rules* that express the interaction model in terms of the predicates defined in the `declare` part.

Like all rules in SCOLL, system rules are denoted as an implication with an (optional) list of preconditions on the left hand side, the implication sign ("=>"), a list of postconditions on the right hand side, and a final delimiter "`;`".

The precondition and postcondition of the rule are the conjunction of the listed predicates.

```
system
1.  access(A,B) access(A,X) B:may.receive()
    A:may.sendTo(B,X)
    => access(B,X) A:did.sendTo(B,X) B:did.receive(X);
2.  access(A,B) access(B,X) A:may.getFrom(B)
    B:may.return(X)
    => access(A,X) A:did.getFrom(B,X) B:did.return(X);
```

The reader can easily verify that these two rules are the formal expression of the two informal rules of our interaction model in section 2.2.4, expressed with the predicates of section 2.2.5.

This particular interaction model will be reused on several occasions in this thesis, but we will also encounter refinements of this model and examples of completely different interaction models.

2.2.7 The `behavior` Part

The behavior of `alice`, `bob`, `caretaker`, and `carol` are modeled here as a set of *behavior rules*, expressed in terms of the knowledge and behavior predicates of section 2.2.5. For the first time we will have to look at the actual code in section 2.2.2.

The behavior of the entities is abstracted from the actual entity to its behavior type. Because of this abstraction, all predicates in this part will be used with their original arity *minus one*. For instance to express that `bob`'s behavior invokes an entity `B` and passes on access to `X`, we will write `may.sendTo(B X)` in `bob`'s behavior type to indicate: `bob:may.sendTo(B X).`

Abstracting the behavior from the entity like this, makes it possible to reuse the same behavior type for different entities and will make it easier to experiment with different behaviors.

In this example every subject has a unique behavior type. Every behavior type has the form:

`<NAME>` "`:`" "{" [rule]* "}".

The name of the behavior type is written in capitals.

The left hand side of a behavior rule is a list of knowledge predicates that indicate the conditions in which an entity with this behavior will cooperate in invocations. The right hand side will indicate the behavior with a list of behavior predicates.

Similar to system rules, the lists on either side of a behavior rule are interpreted as conjunctions and the preconditions can be empty.

```
UNKNOWN:
{ => may.sendTo(A,X) may.getFrom(A) may.return(X)
may.receive();}
```

This will be the behavior for all "unknown" subjects: subjects about which we have no sufficient information about how they are (or will be) programmed. To be safe, we can only assume that these subjects are maximally interactive. They will do anything that the interaction model allows. Therefore the only behavior rule we need is one that derives all behavior predicates from no preconditions.

```
MINIMAL: {}
```

If we are sure that a relied upon entity never interacts with other entities, we can simply model its behavior as an empty set of rules. However, in this case we will use the `MINIMAL` behavior for a different purpose: we will use it as the lower bound for the behavior that can be allowed for `carol`, without violating the safety properties.

```
ALICE: {isBob(B) isCaretaker(C) => may.sendTo(B,C);}
```

Compare this rule with the code for `Alice` in the Oz-E example of section 2.2.2. In the first statement, `{Bob Caretaker _}`, `Alice` invokes `Bob` with the input argument `Caretaker` and ignores any returned value.

The influence of this statement is expressed by this behavior rule: if the entity assumes that `B` is `bob` and that `C` is `caretaker`, it will invoke `B` and pass `C` to it.

The predicates `isBob/2` and `isCaretaker/2` are binary knowledge predates that are *private* to the entity whose behavior is expressed. Private knowledge predicates can be used at will, to express the internal logics of a subject. They form an exception to the rule that every predicate must be declared in the `declare` part. We will meet these predicates again when describing the initial configuration in section 2.2.9.

The second statement, `{Delay 60000}`, has no effect on the propagation of authority and is therefore not modeled. Since SCOLL defines KBMs and KBMs are monotonic approximations, only the authority increasing effects must be modeled.

The third statement, `{Caretaker revoke _}`, is not modeled because the statement's effect is not *increasing* the authority in the model: no access is propagated, only the information (data) represented by the literal `revoke`. Because our simple interaction model does not handle the propagation of information, no behavior in our model can depend on the availability of information to the subject.

For instance if `Caretaker`'s code would generate authority increasing effects that depend only on the availability of information (e.g. the arrival of the `revoke` instruction), we would model that behavior in SCOLL unconditionally.

Since `Alice`'s code cannot have any other authority increasing effects, this single rule models her complete behavior.

```
PROXY: {
        => may.receive();                               (1)
       isCarol(C) => may.getFrom(C);                    (2)
       isCarol(C) did.receive(X) => may.sendTo(C,X);    (3)
       isCarol(C) did.getFrom(C,X) => may.return(X);    (4)
```

Let us compare these four rules with the code for `Caretaker` in the Oz-E example of section 2.2.2.

Since the local variable `Proxying` is never passed outside `Caretaker`'s scope, we don't have to model that variable. The variable has influence on `Caretaker`'s code but we are not required to take that influence into account when safely modeling `caretaker` (the subject) in SCOLL. It is OK to always assume the most collaborative behavior.

Let us move on to the definition of the **`proc`**`{$ In Out}` part. From the mere fact that `In` is used further on in the code, we can already model the first rule for the `caretaker` subject: it will `rCollect()` input unconditionally. In Oz-E the `Out` argument can be used for additional input as well as for output, but that will not change or add anything to the first rule because that rule states already that `caretaker` will unconditionally accept input from its invokers.

The next statement is an **`if .. then ... else... end`** statement which we will safely approximate by ignoring the test and assuming that both outcomes are possible. We can safely ignore the test `In == revoke` because it has no effect.

The first possible outcome, `Proxying :=` **`false`**, will not be modeled because no access is propagated and the influence of the value assigned to the variable `Proxying` is approximated safety by assuming that all values can be assigned to it.

The second possible outcome, `{Carol In Out}` is responsible for the three remaining behavior rules. Since `Caretaker` only invokes the entity known to him as `Carol`, it is OK to use `isCarol(C)` as a precondition for all these rules.

Rule (2) expresses the fact that the `caretaker` will accept return values from the subjects `C` he refers to as `isCarol(C)`. In Oz (and Oz-E), returning a value means that the responder binds a logical variable passed as an argument of the invocation.

Rule (3) expresses the fact that the `caretaker` will forward values he accepted as a responder, to the subjects `C` he refers to as `isCarol(C)`. That models the situation in which either `In` or `Out\` or both are bound by the invoker.

Rule (4) expresses the fact that the `caretaker` will return the values he acquired as return values from invoking subjects `C` he refers to as `isCarol(C)`. That models the situation in which either `In` or `Out\` or both are not bound by the invoker but by `carol`.

Remark In this example we modeled the SCOLL program to safely approximate the existing code in section 2.2.2 to make it easier to explain the meaning of the SCOLL program. In principle, real code is not necessary to write a SCOLL program. All that is needed is an abstract interaction model in which the abstract behavior of the subjects can be expressed. This technique is useful in the design phase when the software engineer wants to reason about the safety in abstract patterns *before* applying them to real code.

2.2.8 The `subject` Part

In this part the subjects are listed and assigned their behavior in the following form: `<subjectName>` ":" `<BehaviorName>`.

Subjects names start with a lowercase letter.

```
subject
    alice:  ALICE
    bob:  UNKNOWN
    caretaker:  PROXY
    ?  carol:  MINIMAL
```

To indicate that we want to maximize `carol`'s behavior, starting from a given minimal behavior "`MINIMAL`", `carol` is preceded by a question mark.

For the SCOLL program to be a safe approximation of the code in section 2.2.2, every possible entity in the software must be modeled as a subject, including the entities created at runtime. Because the subject `bob` has maximal behavior ("`UNKNOWN`"), the default aggregation strategy allows us to model all entities that are created by the `BobModule` or by `Bob`, into the same subject `bob`.

2.2.9 The `config` Part

This part describes the *initial configuration* of the pattern as a list of predicate "facts". Facts are grounded predicates over the subjects declared in the `subject` part.

```
config
    access(alice,alice) access(alice,bob)
    access(alice,caretaker) access(alice,carol)
    access(bob,bob)
    access(caretaker,caretaker)
    access(caretaker,carol)
    access(carol,carol)
    alice:isBob(bob) alice:isCaretaker(caretaker)
    caretaker:isCarol(carol)
```

The access permissions reflect the situation in the the code, except for the additional self-access we assume for every entity. Assuming self-access is not always necessary but it is a safe precaution, certainly when modeling code in object oriented languages that have a *self* pseudo variable.

Because bob is unknown, it is imperative that `bob` is modeled with initial access to himself.

The last two lines in the `config` part assign initial private knowledge facts to `alice` and `caretaker`.

2.2.10 The `goal` Part

This part states the safety properties and the liveness possibilities. They cannot be derived from the code but are inherent to the safety problem we want to solve.

Liveness possibilities are knowledge facts or permission facts, indicating what authority should not be prevented by the restricted behavior of the relied-upon subjects or by the limitations in the initial configuration.

Safety properties are knowledge facts or permission facts, preceded by an exclamation mark indicating what authority should be provably unattainable from the initial configuration.

```
goal
    !access(bob,carol)
```

In this simplified caretaker pattern we have a single safety property stating that `bob` should not get `access` to `carol` because that would render the revocation of the `caretaker` futile.

2.2.11 Applications

SCOLL is naturally suitable for modeling software problems in capability based languages. Capability based languages allow the application of POLA at all levels of detail in software.

But SCOLL is a versatile language and can model all kinds of systems, configurations and problems for the purpose of safety analysis. It is useful for pedagogical purposes because it can formally express patterns and mechanisms of interacting subjects at a suitable level of detail to allow us to reason about the boundaries of the effects (authority) of interactions. It can be used to formally express and compare different strategies for safety enforcement and different styles of protection systems, including ACL based reference monitoring and systems with hybrid protection systems.

Because SCOLL abstracts from the actual interaction model, we also expect to find applications outside the domain of software engineering.

Chapter 6 will describe SCOLL, its syntax and semantics, and its possible extensions.

2.3 SCOLLAR

SCOLLAR is a prototype implementation of SCOLL, based on constraint programming in Mozart/Oz [Sch02]. This section provides an intuition about its purpose in safety analysis.

Detailed instructions for the use of SCOLLAR are presented in chapter 7. SCOLLAR's constraint based design and implementation will be explained in sections 7.6 and 7.7.

2.3.1 Purpose

SCOLLAR is a tool to analyze safety in patterns of collaborating subjects expressed in SCOLL. It can be used for several purposes:

1. To check if a given pattern guarantees a set of safety properties without necessarily preventing another set of liveness possibilities.

   For instance, subject `bob` should never get access to subject `carol` (safety), but there should at least be one possible scenario in which `carol` gets access to `bob` (liveness possibility) .

2. To search for (all) safe ways to restrict the interaction between the subjects in the pattern, such that:

   - No safety property is violated
   - No liveness possibility is prevented
   - Every set of restrictions (solution) is minimal for safety: adding a restriction is not necessary and removing a restriction will break at least one safety property.

   The user decides what kind of restrictions can be imposed. Typically, the imposed restrictions will affect the behavior of the subjects the user can rely on or control and/or part of the initial configuration of the pattern (e.g.: part of the initial permissions or the initial state of the subjects).

   Using SCOLLAR in this way is most useful when designing/programming secure patterns of interaction between some trusted (relied upon) and some untrusted (unknown) subjects.

2.3.2 Example

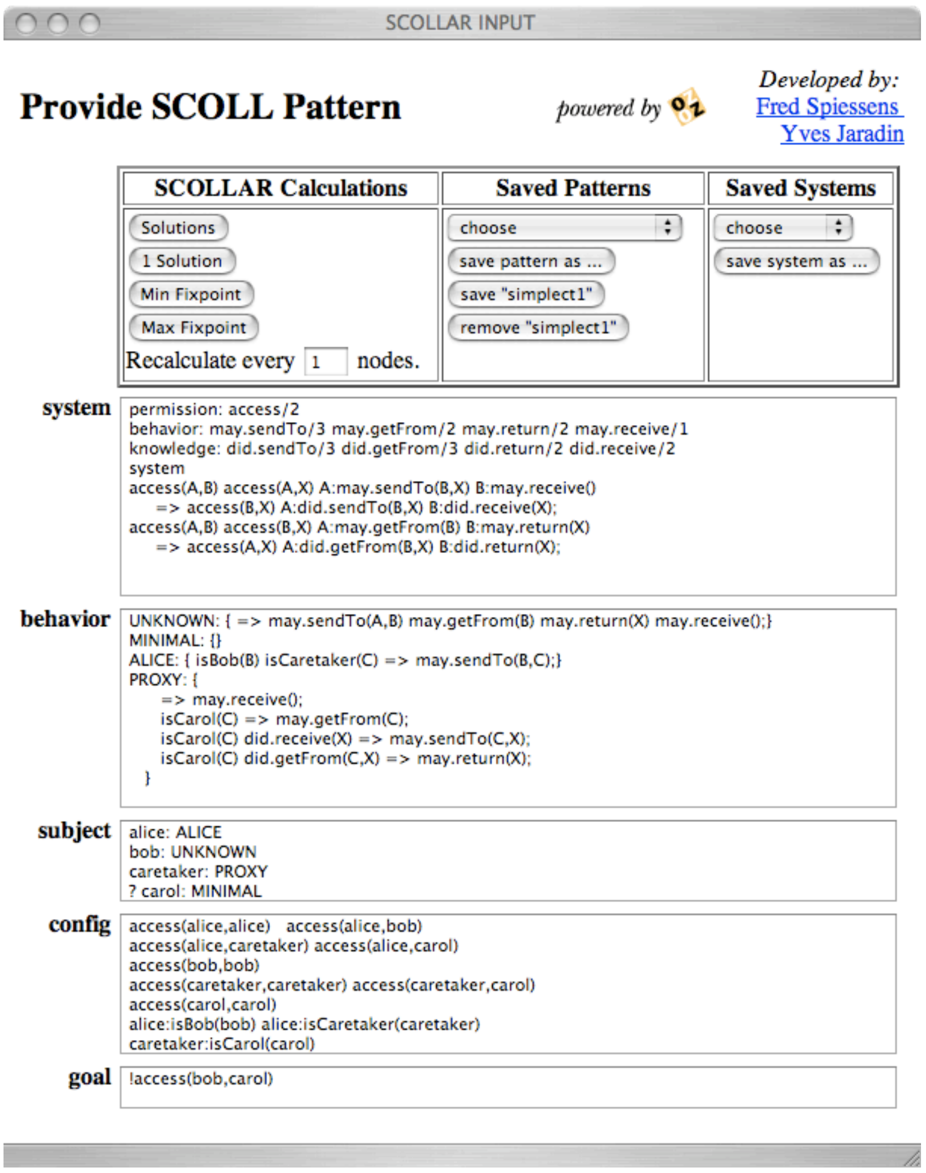
Figure 2.4 shows the web-based user interface of SCOLLAR, with the SCOLL program we specified in section 2.2.3. A push on the "solutions" button instructs SCOLLAR to calculate the necessary restrictions in `carol`'s behavior.

Figure 2.5 shows the two solutions to `carol`'s behavior, found by SCOLLAR. The solution is presented as an access graph, generated in GraphViz [GV05], and a table that gives an overview of the solutions.

The graph will only be generated when `access/2` is declared as a binary permission in the SCOLL pattern. Future versions of SCOLLAR will allow the user to choose the binary relations to be visualized. The solid arrows indicate access that was present in the (minimal) initial configuration. The dashed arrows represent access that is attainable in all solutions. The dotted arrows indicate access that is attainable it at least one solution but not in all.

The table shows the solutions in the columns. In the rows it lists `carol`'s behavior facts that are illegal in at least one solution. Behavior that is to be prevented is indicated with "*0*".

In both solutions the last row indicates that `carol` should not return herself when being invoked (`may.return(carol carol)`. Solution 1 adds the restriction that `carol` should not invoke `bob` with herself as input argument (`may.sendTo(carol bob carol)`.



system

```
permission: access/2
behavior: may.sendTo/3 may.getFrom/2 may.return/2 may.receive/1
knowledge: did.sendTo/3 did.getFrom/3 did.return/2 did.receive/2
system
access(A,B) access(A,X) A:may.sendTo(B,X) B:may.receive()
    => access(B,X) A:did.sendTo(B,X) B:did.receive(X);
access(A,B) access(B,X) A:may.getFrom(B) B:may.return(X)
    => access(A,X) A:did.getFrom(B,X) B:did.return(X);
```

behavior

```
UNKNOWN: { => may.sendTo(A,B) may.getFrom(B) may.return(X) may.receive();}
MINIMAL: {}
ALICE: { isBob(B) isCaretaker(C) => may.sendTo(B,C);}
PROXY: {
     => may.receive();
     isCarol(C) => may.getFrom(C);
     isCarol(C) did.receive(X) => may.sendTo(C,X);
     isCarol(C) did.getFrom(C,X) => may.return(X);
  }
```

subject

```
alice: ALICE
bob: UNKNOWN
caretaker: PROXY
? carol: MINIMAL
```

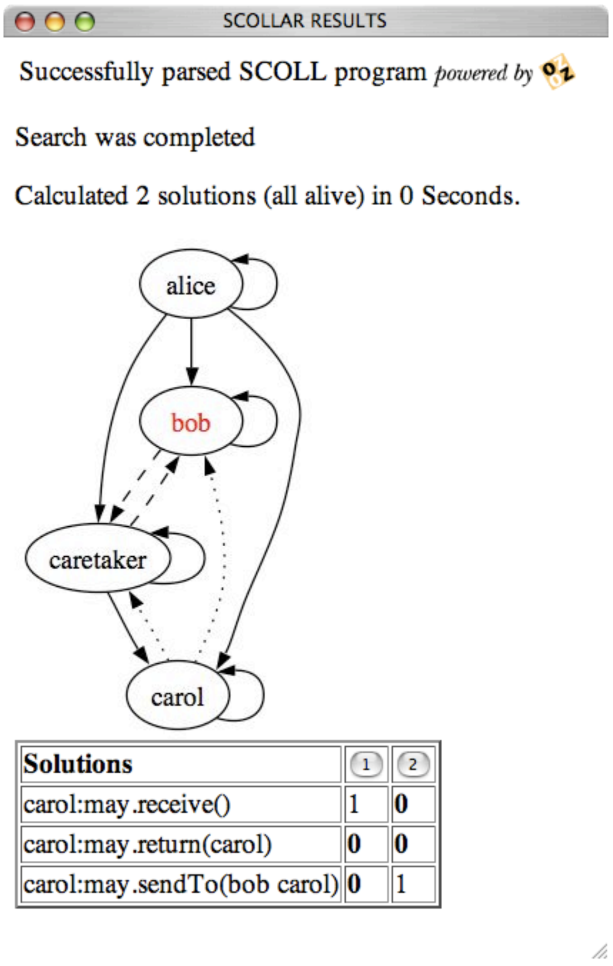
config

```
access(alice,alice)   access(alice,bob)
access(alice,caretaker) access(alice,carol)
access(bob,bob)
access(caretaker,caretaker) access(caretaker,carol)
access(carol,carol)
alice:isBob(bob) alice:isCaretaker(caretaker)
caretaker:isCarol(carol)
```

goal

```
!access(bob,carol)
```

Figure 2.4: The simple caretaker example in SCOLLAR


SCOLLAR RESULTS

Successfully parsed SCOLL program *powered by*

Search was completed

Calculated 2 solutions (all alive) in 0 Seconds.

Solutions	1	2
carol:may.receive()	1	**0**
carol:may.return(carol)	**0**	**0**
carol:may.sendTo(bob carol)	**0**	1

Figure 2.5: Overview of the solutions in SCOLLAR

Solution 2 restricts `carol` from accepting input `may.return(carol,carol)`. That will prevent her from getting access to `bob` and therefore her invoking behavior towards `bob` needs no restrictions.

The top row of the table contains a button for every solution. Clicking on that button will reveal a page that shows the complete details of the solution:

- The access graph corresponding to the maximal propagation of access for the solution.
- A table for every subject, listing that subject's permissions, knowledge, and behavior facts, and indicating for each of them whether they are attained, not-attainable, or irrelevant in the solution.

Chapter 7 will explain in detail how the solutions should be interpreted and what other options SCOLLAR provides. Chapter 8 contains a set of elaborated examples of safety analysis in SCOLLAR, of which the solutions are show and discussed.

Part I

Foundations

Chapter 3

Formal Systems for Safety Analysis

This chapter reviews two existing and well known formal protection models that have had a major influence in the domain of security analysis. They are presented here because they form the basis of the new formalism called "Knowledge Behavior Models", described in chapter 5.

The first of them, named "Formal Protection Systems", was presented in 1976 by Harrison, Ruzzo, and Ullman [HRU76]. Formal Protection Systems are very expressive and were used to prove that the safety question in general – whether a certain right can get into the "wrong" hands in a certain situation – is not computable.

On the other side, less concerned with expressive power than with tractability, was the research on tractable formal systems in which the safety properties are computable in polynomial time. One such system was presented in the same year by Jones, Lipton and Snyder [JLS76], and has safety properties that can be computed in time linear to the size of the problem. The second formalism we discuss in this chapter is an extension of this system, proposed in 1979 by Bishop and Snyder, and called "Take-Grant Systems" [BS79].

The original Take-Grant systems are not well suited for practical safety analysis. Many modifications, extensions, and alternatives were proposed [Bis81, San88, FB96]. None of these *explicitly* explored the notions of behavior and collaboration, even though we can discover the embryonic form of these notions in the original Take-Grant systems.

We will rediscover the notions of behavior and collaboration in chapter 4 where we present capability-based security. We will develop these notions into explicit formal concepts in chapter 5 where they will play a central role in our new practical formalism called "Knowledge Behavior Models".

3.1 Formal Protection Systems

We will refer to formal protection systems as HRU systems or simply HRU.

3.1.1 Introduction

The protection state of a system is modeled as a two-dimensional matrix. The rows of that matrix represent subjects (entities that can have rights), and the columns represent objects (entities a subject can have rights on). The cells of the matrix contain the actual rights the subject has on the object.

Possible transitions in the protection state are conditional on the rights that are present in the protection state. The following section defines the formal concepts.

3.1.2 Definitions

Definition 5. *HRU Protection System*

A *protection system*, as defined in [HRU76], is a couple $\langle C, R \rangle$ in which:

- C is a finite set of commands.
- R is a finite set of generic rights (e.g. *read, write, grant,* ...).

All rights in R are binary: a subject can have rights from R on an object. For instance if the right *read* is present in a cell of the table where the subject in the row is *alice* and the object in the column is *rootDirectory*, that means: *alice* has the right to read the *rootDirectory*.

The commands in C each describe a set of similar transitions between configurations, which will be defined hereafter. The commands are expressed using a series of formal arguments (variables) of which the actual values represent the subjects and/or objects that are relevant in the state transition.

Definition 6. *HRU Configuration*

A configuration is a tuple $\langle S, O, P \rangle$ in which:

- S is a countable set of subjects (modeling entities that can have rights).
- O is a countable superset of S containing objects (the entities subjects can have rights on).
- P is a matrix in which the rows represent the subjects in S and the columns represent the objects in O. At the intersection $P(s, o)$ of row s and column o, the matrix P stores the set of rights s has on o.

  Formally, P is a function from $S \times O \rightarrow 2^R$.

HRU Commands

The commands of a protection system are expressed with formal parameters in the form:

```
command $c(X_1, X_2, \ldots, X_n)$
if $r_1 \in P(X_{i_1}, X_{i_2})$
    $\ldots$
    $r_k \in P(X_{j_1}, X_{j_2})$
then $operation_1$
        $\ldots$
        operation_m
    end
```

When a command is applied, some of its actual arguments will range over O, while others can only be in S.

The part between `if` and `then` is optional. It contains a list of explicit preconditions for the command to be applicable, in the form: $r \in P(X_1, X_2)$, requiring the presence of r in the cell at row X_1 and column X_2 in the matrix.

The remaining part between `then` and `end` is called the command's *body*, and consists of a sequence of primitive operations that are to be applied in order and atomically (without interference of other commands). Operations represent basic transformations to a configuration.

Both parts can also contain implicit preconditions. A reference to a cell $P(X_1, X_2)$ in either of the parts expresses the implicit preconditions that $X_1 \in S$ and $X_2 \in S \cup O$. The operation-specific implicit preconditions are listed with the operations in table 3.1.

Operations : There are six kinds of operations. We present them below as transitions $Q_i \Rightarrow_{op_i} Q_{i+1}$ from a configuration $Q_i = \langle S, O, P\rangle$ to $Q_{i+1} = \langle S', O', P'\rangle$, with their preconditions on Q_i and their effects in Q_{i+1}.

Table 3.1: Primitive operations on HRU configurations

operation	conditions	effects
`enter` r `into` $P(s,o)$	$r \in R$ $s \in S$ $o \in O$	$Q_{i+1} = \langle S, O, P'\rangle$ $P'(s,o) = P(s,o) \cup \{r\}$ $(x,y) \neq (s,o) \Rightarrow P'(x,y) = P(x,y)$
`delete` r `from` $P(s,o)$	$r \in R$ $s \in S$ $o \in O$	$Q_{i+1} = \langle S, O, P'\rangle$ $P'(s,o) = P(s,o) \setminus \{r\}$ $(x,y) \neq (s,o) \Rightarrow P'(x,y) = P(x,y)$
`create object` o	$o \notin O$	$Q_{i+1} = \langle S, O \cup \{o\}, P'\rangle$ $\forall x \in S : P'(x,o) = \{\}$ $\forall (x,y) \in S \times O : P'(x,y) = P(x,y)$
`create subject` s	$s \notin O$	$Q_{i+1} = \langle S \cup \{s\}, O \cup \{s\}, P'\rangle$ $\forall (x,y) \in (\{s\} \times O) \cup (S \times \{s\}) :$ $P'(x,y) = \{\}$ $\forall (x,y) \in S \times O : P'(x,y) = P(x,y)$
`destroy object` o	$o \in O \setminus S$	$Q_{i+1} = \langle S, O \setminus \{o\}, P'\rangle$ $P' = P\|_{S \times (O \setminus \{o\})}$ (P restricted to domain $S \times (O \setminus \{o\})$)
`destroy subject` s	$s \in S$	$Q_{i+1} = \langle S \setminus \{s\}, O \setminus \{s\}, P'\rangle$ $P' = P\|_{(S \setminus \{s\}) \times (O \setminus \{s\})}$

The operations in a command specify additional implicit preconditions (second column in table 3.1) on S and O in the configuration Q_i. Adding or deleting a right requires the existence of the subject in S and of the object in O. The creation of subjects (objects) requires that the subject (object) does not yet exist in S (O). Destruction operations require previous existence in S or O.

Adding a right to $P(s, o)$ does not require the right to be absent from $P(s, o)$ before: in that case the operation simply has no effect. Dropping a right is always possible too and has no effect if the right was absent.

Combining all these preconditions, a command α can be executed in configuration Q if :

- Every precondition expressed between `if` and `then` in α applies in Q, and
- $\forall i \in \{i \dots n\}$ with n being the number of operations in α
  and op_i being the i-th operation in α:
  The (implicit) preconditions for op_i apply in Q_i such that
  $Q = Q_1$ and $\forall i \in \{i \dots n\} : Q_i \Rightarrow_{op_i} Q_{i+1}$

To decide whether or not a configuration can evolve into a certain (unsafe) state, the intermediate states reached by executing the individual operations of a command in turn will be taken into account. Commands that introduce a right in one operation and remove it in the next one are recognized as having introduced the right.

If more than one command is ready to be executed, any one of them can be chosen non deterministically.

Definition 7. *A Step $Q \vdash_\alpha Q'$*

If a configuration Q contains a series of entities (subjects and objects) onto which a command α that contains n operations op_i can be applied, the configuration resulting from executing the command is:

$$Q' = Q_n \text{ such that } Q = Q_0 \text{ and } Q_0 \Rightarrow_{op_1} Q_1 \Rightarrow_{op_2} \ldots \Rightarrow_{op_n} Q_n.$$

We write $Q \vdash_\alpha Q'$. When α is not relevant, this is abbreviated to : $Q \vdash Q'$. The transitive, reflexive closure of $\vdash$ is denoted $\vdash^*$.

3.1.3 The Safety Problem

A configuration is safe for a certain right $r \in R$ if this right cannot be introduced into the access matrix (*leaked)* by any operation inside any command that can become executable after a finite number of steps $\vdash$. To express more specific concerns that consider also object and/or subject roles (e.g. what object is the right pointing at or what subject is the right leaked to) a purpose-built protection system and configuration must be *derived* from the original one, in which the concern can be expressed as a simple leakage problem.

Definition 8. *Leakage:*
Given an arbitrary protection system $\langle C, R \rangle$,
an arbitrary configuration $Q = \langle S, O, P \rangle$,
and an arbitrary right $r \in R$
Q leaks r $\iff$ *$\exists$ command $\alpha \in C$ and $\exists\, Q', Q'' : Q' = \langle S', O', P' \rangle$*
and $Q'' = \langle S', O', P'' \rangle$
$Q \vdash^* Q' \vdash_\alpha Q''$
α's body = $op_1 \ldots op_n$
α is applied to the actual arguments $X_1 \ldots X_m$
$\exists\, Q_1 \ldots Q_n : Q' = Q_0 \Rightarrow_{op_1} Q_1 \Rightarrow_{op_2} \ldots \Rightarrow_{op_n} Q_n = Q''$.
At least one operation op_i in α is: `enter` r `into` $P(X_j, X_k)$
such that :
$Q_{i-1} = \langle S_{i-1}, O_{i-1}, P_{i-1} \rangle$, $Q_i = \langle S_{i-1}, O_{i-1}, P_i \rangle$, and $r \notin P_{i-1}(X_j, X_k)$

Notice that the resulting configuration Q'' may *not* have an extra right r in its matrix. In that case the `enter` operation in α must have been followed by a complementary `delete` or `destroy` operation in the same command, that removed the trace from the final result Q''. As a matter of fact, Q'' may very well be identical to Q' or even to the original Q.

Rights Exertion versus Delegation

Commands that perform only such “encapsulated” leaks effectively prevent other commands from making use of the leaked rights. Following [HRU76] this is useful to model the *exertion* of a right: e.g. calling a $write$ procedure on a file gives the caller temporary write-access to the file but the right remains encapsulated in the procedure call and cannot be used by other commands. Inter-command leakage could then be reserved to model rights *delegation*.

Our main objection to this interpretation of internal leakage is that it is not compositional. How would we model a call to a procedure that calls the $write$ procedure? How would we model an arbitrary long chain of calls ending in a call of the $write$

procedure? Using the intra-command leaking approach we would have to compose the operations of the commands that model a single call into new commands: one for every possible chain of calls. Even with a finite set of subjects, each modeling a procedural closure, this could easily lead to an infinite number of commands, which is not allowed in the definition of protection systems.

We could consider using an extra right $exerted\_r$ to indicate that a right r was exerted and replace all intra-command leakage of r by inter-command leakage of $exerted\_r$. This is a compositional way to model right exertion and it renders intra-command leaks superfluous for this purpose. The set of rights R is still finite: it would only double in size.

The alternative approach to model rights exertion does not remove the need to *detect* intra-command leaks. When an `enter` operation is followed by a `destroy` of the subject or the object involved in the leak, the $exerted\_r$ right can no longer be detected in the resulting configuration. This poses no problem for the composed right exertion though: `destroy` could model a procedure and its closure going "out of scope" before the chain of calls ends.

This problem shows already that modeling actual safety problems in HRU systems can be hard, even if the formalism itself is Turing Complete (Section 3.1.4).

Converting Complex Safety Problems

To consider the leakage of a right r pointing to a particular object $x \in O$, [HRU76] suggests deriving the protection system $\langle C', R' \rangle$ from $\langle C, R \rangle$ and the configuration $Q' = \langle S', O, P' \rangle$ from $Q = \langle S, O, P \rangle$, such that:

$R' = R \uplus \{r_x, r_{leak}\}$ (create 2 extra rights)

$C' = C \cup \{c_{dum}\}$ with

```
command $c_{dum}(X_1, X_2)$
if $r \in P(X_1, X_2)$
    $r_x \in P(X_2, X_2)$
then enter $r_{leak}$ into $(X_1, X_2)$
end
```

$S' = S \cup \{x\}$ (turn x into a subject)

$P'(x, x) = \{r_x\}$ (r_x is the right that "identifies" x)

In the derived system and configuration, x is identified by its unique right r_x to itself that is never leaked. Leaking r-to-object-x in Q is converted into leaking r_{leak} in Q' by the new command c_{dum}.

To consider a single subject the right should not be leaked to, [HRU76] proposes a similar trick. The subject is again given a unique right to identify itself and the new r_{leak} will only be leaked if r is leaked to the subject. The approach can be extended to sets of subjects (and/or objects) by giving them all a right that identifies the set.

The attentive reader may have noticed a problem with the suggested approach: it does not detect intra-command leakage of r ! The right way to convert is to add n duplicates of every command α in C that has an `enter` operation for r, n being the number of object arguments of α. The i-th duplicate of this command should test if the i-th object argument can be identified as x, and add the `enter` operation for r_{leak} for every operation that leaks r to this object.

$C' = C \cup \bigcup_{c \in C_r} \{c_1 \dots c_n\}$ with
C_r being the subset of commands that enter r, and
n being the number of object arguments in c, and
c_i being the i-th duplicate of c adapted in this way:

```
command $c_i(\dots, X_{o_1}, \dots, X_{o_i}, \dots, X_{o_n}, \dots)$
if ...
   $r_x \in P(X_{o_i}, X_{o_i})$ (extra condition)
then ...
     enter $r$ into $(X_j, X_{o_i})$
     enter $r_{leak}$ into $(X_j, X_{o_i})$ (extra operation)
     ...
end
```

Another problem remains with this approach. Because x is turned into a subject, some of the commands in C that were not applicable in Q because of (implicit) pre-conditions on x being a subject, can now become applicable in Q'.

The original paper ([HRU76]) gave a broad and general introduction to the formalism and its most important result was the undecidability of the safety problem (next section). The paper has left some details formally unspecified and contains more little flaws. These flaws are not hard to detect upon a second reading of the paper. None of them affect the main contribution of the paper.

3.1.4 Is Safety Computable?

HRU models were not devised to perform practical safety analysis, but to reason about the computational limits of safety properties. The main result of [HRU76] is the undecidability of the safety problem: there is no algorithm to decide in general (for all HRU systems and configurations) if a certain right (on a certain object) will leak (to a certain subject).

This result is proven by constructing a protection system that simulates an arbitrary Turing machine such that the leakage of the right r corresponds to the Turing machine entering a finite state, a condition that is known to be undecidable [Tur37].

Because the work of Harrison, Ruzzo, and Ullman does not consider any form of (semantic) equivalence relation between protection systems and/or configurations, it cannot pose the more relevant questions:

- Is there for every protection system $\langle C, R \rangle$ (and configuration $\langle S, O, P \rangle$) a protection system $\langle C', R' \rangle$ (and configuration $\langle S', O', P' \rangle$) for which the safety problems are decidable, and that has the same protection results?
- Is there an algorithm to compute such a $\langle C', R' \rangle$ (and $\langle S', O', P' \rangle$) in case one exists?

3.1.5 Relying on Subjects in HRU Systems

We have indicated in the previous sections that protection systems are not devised (and not fit) for practical safety analysis because of the awkward way precise safety properties have to be defined.

This becomes even more obvious when we want to consider the influence of the behavior of the modeled entities on the safety in such systems. Two types of behavior can be modelled straightforward.

- Completely untrusted entities can be modelled as subjects: they will exert every right the commands allow.
- Completely passive entities (not exerting rights or propagating rights) can be modelled as objects: they will never have a right themselves.

To mimic the restricted influence of subjects that are relied upon to restrict their behavior, [HRU76] proposes to simply delete the relied upon subjects from the configuration. Doing so will indeed answer the question: "*Can a certain right r leak without the cooperation of the relied upon subject(s)?*" but it does not allow us to ask: "*What level of cooperation would be allowed*?". It is all or nothing.

The paper also overlooks the fact that simply deleting a relied-upon subject could enable another command that tries to create the subject but was prevented from doing so in the original configuration because of the implicit precondition of the `create subject` command.

The drastic all-or-nothing approach leaves no room to directly model more refined restricted behavior of entities that do exert some rights in certain circumstances but not any other rights in any other circumstances. To model more fine grained behavior we have to build it into the commands and there is no preferred or "natural" way to do this.

3.1.6 Discussion

The undecidability of the safety problem is an important result. If we are going to model real safety problems in a formalism that is decidable, we must choose between the following two options:

1. Restrict our domain of interest to decidable problems and calculate their solutions.
2. Keep our original domain of interest but calculate safe but approximate solutions, never indicating *safe* when the actual problem can be *unsafe*, but possibly, due to the approximation, resulting in a falsely unsafe estimate.

In this thesis we choose the second option. We want to be able to model a safe and decidable approximation for every conceivable safety problem. Therefore the results in this work apply to all kinds of safety problems. The accuracy of our model, its ability to avoid false unsafe estimates, will be easily adaptable.

Because of its decidable nature the model will not be Turing complete. Instead, we will give priority to its practical utility for software designers and developers to accurately express safe approximations of the safety problems they encounter in real software.

3.2 Take-Grant systems

In this section we introduce the Take-Grant systems as they are described in [BS79], including the concepts that were introduced in that work to model the propagation of *de-facto* authority: authority that is not tracked or reflected in the permissions of the protection state.

3.2.1 Protection Graphs

Take-Grant systems are configurations of subjects and objects, propagating, manipulating, and using rights following a predefined, fixed set of rules. They are represented as a directed graph called the "protection graph", consisting of nodes and labeled solid arcs. The nodes are either subjects or objects. Contrary to protection systems (Section 3.1), objects can have rights too: they just cannot use them. We will therefore no longer refer to them as objects but as passive subjects. We will refer to subjects as active subjects and use the term subject for both active and passive subjects.

Rights are represented by labels on solid arcs in the graph from the subject having the right to the subject the right can be applied to. The combination of a right with the subject it can be applied to is called a capability. An arc labelled with n different rights thus represents n different capabilities. Take-Grant systems use capabilities as the atomic unit of rights manipulation. We call the origin of the arc the holder of the capability and the end of the arc its target.

In [BS79] the authors discuss the transfer (propagation) of information as well as of authority. Therefore they propose to consider that a subject can have two kinds of authority:

1. The authority that is "*formally recorded in the protection system"*, meaning : the authority that corresponds directly to the holder's set of capabilities. This authority is referred to as *de-jure* authority and is represented in the graph by solid arcs labeled with one or more rights. A solid arc pointing to a subject is called a capability as it combines a designation (to the subject) with a permission to use that subject in a certain way.

2. The authority that is not formally recorded in the protection system. This authority is called *de-facto* authority and is represented in the graph by dashed arcs that are labeled with one or more rights. It indicates that a subject is able to reach a goal similar to the authority provided by a *de-jure* right. For instance *de-facto read*-authority could be attained "… *(usually in the form of a copy and with the assistance of others), without necessarily being able to get the direct authority to read the information.*"

The actual set of *de-jure* and *de-facto* rights is not fixed but can be modeled to suit the situation. Table 3.2 shows the rights that are used for the explanation.

Table 3.2: Rights in Take-Grant Systems.

$read$	right to *read* the contents of (get data from) a subject
$write$	right to *write* (provide data) to a subject
$take$	right to *get capabilities* from a subject
$grant$	right to *give capabilities* to a subject

In the following representation, active subjects are depicted as black nodes, passive subjects as white nodes, and subjects in general as grey nodes. Since active subjects can always play the role of passive subjects, we will only need black and grey nodes to explain the rules for transfer of information and authority. Arcs representing *de-jure* rights are depicted with solid lines. Arcs representing *de-facto* rights as dashed lines.

3.2.2 De-Jure Rules

De-jure rules govern the transfer of *de-jure* rights, indicated with solid arcs. They all respect the atomic nature of capabilities when manipulating rights, as they never dissociate a right from its target. They also respect the principle of attenuation, as rights can be transferred only by duplicating capabilities. New capabilities are only created when a new subject is created.

The *de-jure* rules are depicted graphically in figure 3.1 with their preconditions (left) and effects (right). Labels with capital letters denote sets of rights.

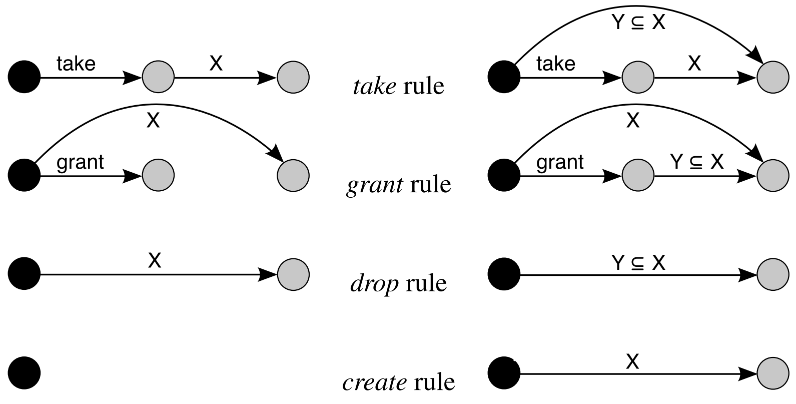


Figure 3.1: *de-jure* rules

Figure 3.1 shows how rights are propagated (take and grant rules) by copying (a subset of) the available capabilities. The preconditions always involve one black node: the active subject that is “responsible” for the effect of the rule. Here is a short explanation of what the rules do.

Take A *take* capability allows the owner to copy the target subject’s capabilities to himself, whether that target subject is active or passive.

Grant A *grant* capability allows the owner to copy his capabilities to the target subject, whether that subject is active or passive.

Drop Every active subject can always drop a subset of its capabilities. When the last capability towards a subject is dropped, the arc itself is removed. No special right is necessary for dropping but only active subjects can drop capabilities. Of course, subjects can only drop their own capabilities.

As all propagation in take-grant systems only depends on the presence of rights and capabilities – never on the absence of a right or a capability – dropping rights or capabilities cannot lead to more propagation. This means that when calculating safety properties (upper bounds to the propagation) there is no need to consider the possible dropping of capabilities.

Create Active subjects can create new subjects (either passive or active) and thereby get all possible capabilities to that subject.

The design of the *de-jure* rules makes the formalism already much more practical for our purpose of modeling behavior for the following reasons:

- The rules naturally model the behavior of programmed entities: every change to the configuration (every rule application) is “performed” by some active subject.
- The preconditions naturally model restrictions in terms of “what a subject is able to do” given its capabilities.
- The principle of attenuation is guaranteed by the rules: “no subject can delegate rights (capabilities) it does not have”.

Keep in mind that this set of *de-jure* rules is only meant as an example set. More elaborate rules can be modeled in the same style, refining the conditions in which *de-jure* transfer of authority takes place.

3.2.3 De-Facto Rules

The effects of using *take* and *grant* capabilities are modeled by the *de-jure* rules for *take* and *grant* rights. Using *read* and *write* capabilities does not result in a transfer of *de-jure* authority. Nevertheless, the fact that information is propagated implies that this information may become reachable to other entities, and thus that *de-facto* authority is propagated.

The *de-facto* rules indicate how *de-facto* authority to read and write information can propagate. They are depicted graphically in figure 3.2 with their preconditions (left) and effects (right).

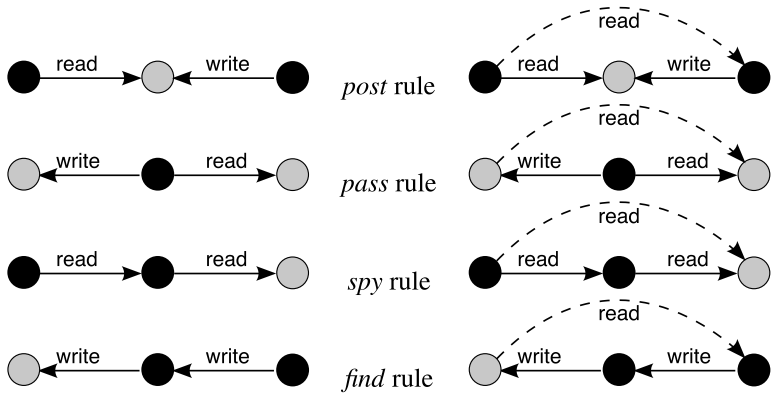


Figure 3.2: *de-facto* rules

Post An active subject (left) that can read a subject (middle) that can be written to by a third active subject (right), has *de-facto read* authority to the latter.

Pass A subject (left) that can be written to by an active subject (middle) that can also read from a third subject (right), has *de-facto read* authority to the latter.

Spy An active subject (left) that can read from an active subject (middle) that can read from a third subject (right), has *de-facto read* authority to the latter.

Find A subject (left) that can be written to by an active subject (middle) that can at its turn be written to by another active subject (right), has de-facto *read* authority to the latter.

Keep in mind that this set of *de-facto* rules is only meant as an example set. More elaborate rules can be modeled in the same style, refining the conditions in which *de-facto* transfer of authority takes place.

From these *de-jure* and *de-facto* rules we make the following observations about the relation between authority and permissions in Take-Grant systems:

1. Subjects that do not have permissions can have authority, as is shown in the *pass* and *find* rules.

2. Subjects that do not *use* their permissions can still have authority, as is shown by the same *pass* and *find* rules.

3. Subjects that do not use their permissions still enable the propagation of *de-jure* and *de-facto* authority by allowing active subjects to use them as a communication channel for capabilities and information.

4. The authority a subject can have by using its permissions is only partially defined by the permission itself. The other part is defined by the type of subject the permission is applied to: active or passive.

Notice that, given the proposed set of *de-jure* and *de-facto* rules, two subgraphs can only remain authority-separated (kept from influencing each other) as long as they are connected only via paths that have at least two consecutive passive subjects. This renders Take-Grant systems less practical for modeling safety problems in software engineering than we would have hoped. For instance, to model a software entity that can be used as a communication channel for data but not for capabilities, we would have to model the entity as a subgraph in the Take-Grant systems.

A refinement for this situation has been proposed as "Schematic Protection Models" [San88] in which arbitrarily many (static) types of subjects can be constructed whose reducing effects on the transfer of authority is modeled explicitly (see section 11.1.1) .

Observation 4 in the list above is very important if we want to rely on restricted subjects to reduce the authority that unrestricted (untrusted) subjects can have. In collaborative systems this reduction will be complete: permissions will only provide the authority to *try* to attain an effect by collaborating with the target of the capability, who will then decide if and when the intended authority will be realized.

A detailed discussion of the relation between permissions and authority will be given in section 3.4.

3.2.4 Safety Analysis

Safety properties in a Take-Grant configuration graph G are expressed using a predicate *canKnow(p,q,G)*. The predicate expresses that subject p can get the authority to get information from subject q via a finite series of *de-jure* and/or *de-facto* authority transfers.

Algorithms to check safety properties are proposed in [LS77] and [FB96]. The tractability is due to the fact that a single generation of created subjects (just one newly created active subject for every active subject in the initial graph) is enough to enable the maximum propagation of capabilities and data. The details of this calculation and of the tractability proof are omitted as they have no relevance for the further development of the concepts and models in this thesis.

3.2.5 Safe Approximations of Information Propagation

Take-Grant systems introduced *de-facto* rules to analyze the propagation of information. Because in most actual software systems untrusted entities can also use covert channels [Lam73] to propagate information, the actual propagation of information cannot be safely approximated in the Take-Grant model.

In the new approach we will present in Chapter 5, we will model propagation of information for a different purpose: to inform the relied-upon entities about the effects of legal exertions of permissions their behavior depends upon. That will be safe because:

- Untrusted subjects will be modeled not to rely upon this information anyway.

  For instance, in the example of figure 2.3 in section 2.2.3 `UNKNOWN` behavior does not depend on any conditions.

- We rely upon the protection system in the software to make sure that the relied-upon entities in the software cannot be convinced that they were not involved in a legal exertion of permissions, if that was actually the case.

  In the same example, the `caretaker` (with behavior: `PROXY`) is guaranteed to receive the knowledge `did.receive(X)` for every subject X it can possibly accept as an input argument when being invoked.

3.3 Discussion and Comparison

Take-Grant systems contain both interesting restrictions and useful additions to HRU. In this section we discuss the features that are relevant for our purpose of building an expressive formal model for the propagation of authority that can take behavior into account.

3.3.1 Local Preconditions govern the Propagation of Authority

The *de-jure* and *de-facto* rules in Take-Grant systems involve only preconditions that directly concern the capabilities and the authority of the subjects that are involved in the transfer of authority:

1. Appropriate permissions should exist between the subject that delivers the authority and the subject that acquires it. At least one of them must have a capability that targets the other one.

2. The principle of attenuation: the subject that delivers authority can only deliver the authority that it holds itself.

We can identify the following pairs of complementary roles for the subjects:

Initiator and Responder : There is always an active subject (black node) that holds a capability and there is always the not-necessarily-active subject that is the target of this capability (grey node). Let us call the former the *initiator* of and the latter the *responder*. Initiator and responder correspond to the holder and target of a capability.

Emitter and Collector : One of the subjects provides authority to the other: the first one we call the *emitter* of the authority, the other one is the *collector*.

In a *grant* or *write* situation the initiator is the emitter and the responder is the collector. In a *take* or *read* situation the initiator is the collector and the responder is the emitter. These two pairs of complementary roles allow us to model the interaction in many actual programming languages.

For instance in a pure functional programming paradigm, access to an entity X can be passed from entity A to entity B in exactly these two ways:

1. A invokes B with X as an input argument: A is the initiator and emitter whereas B is the responder and collector.

2. B invokes A who returns X: B is the initiator and collector whereas A is the responder and emitter.

Actual invocations can be composed of these atomic interactions, for instance when A invokes B with an input argument X and B returns Y to that invocation.

The complimentary roles are relevant for many other programming paradigms too. For instance in object oriented programming, message passing can be modeled using the same atomic patterns of interaction, with the sender as the initiator. Even the binding of a logical variable to a value and the assignment to a mutable variable or cell can all be modeled in this way: the variable being a "passive" entity that is ready to collect, emit, and respond in an interaction but not to initiate an interaction itself.

We will use the terms *invoker* and *initiator* interchangeably in this thesis.

3.3.2 Modeling Authority, not just Permissions

The most important difference between the two formal systems is the fact that Take-Grant systems detect a form of authority that does not correspond to a single permission. For instance, the *spy* rule shows that in a configuration where `bob` has *read* permission to `carol`, `alice` can use her *read* permission to `bob` to the same effect as if she would have had *read* permission to `carol` directly.

As shown in figure 3.3, when `alice` has no *de-jure* authority to read `carol`, (only) `bob` can make the difference: when `bob` is passive he does not (or cannot) use his permission to read `carol` and therefore `alice` will not have the authority to do so either.

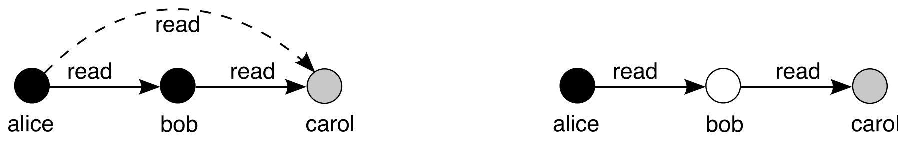


Figure 3.3: The behavior of `bob` decides if `alice` has *read*-authority to `carol`.

Notice that the only *de-facto* authority used in [BS79] is *read* authority but there is no reason why *de-facto* authority should be restricted to the authority to access or transfer data. All sorts of authority can propagate without propagation of permissions: the authority to use an encryption/decryption service, to change a pixel color on the screen, to use any kind of resource, including memory, CPU time, Internet access, all this power can become available to a subject without the need to propagate a single permission, only by relying on other subjects that do have relevant permissions to interact in a suitable way.

Such indirect forms of authority can only be modeled in HRU systems if the HRU protection matrix includes all effects that are relevant for authority.

3.3.3 Modeling Static Behavior

Both HRU and Take-Grant systems consider two kinds of entities. HRU differentiates subjects from objects. Take-Grant systems differentiate between active subjects and passive subjects. Extra (implicit) preconditions in the HRU commands make sure that non-subject objects never get any rights. Extra preconditions in the Take-Grant rules (the black nodes) ensure that only active subjects can use their permissions.

Both types can be used to model entities with relied-upon restricted behavior. HRU non-subject objects can model entities with no behavior (completely uncooperative). Take-Grant passive subjects can model entities that do not use their permissions but always cooperate when permissions are used on them. Whereas HRU non-subject objects can never have rights, passive subjects in Take-Grant systems can get permissions and act as a channel to pass permissions and information between two active subjects.

Schematic Protection Models [San88] extend Take-Grant systems in the a way that allows us to express all kinds of static behavior. Section 11.1.1 discusses the relation between KBMs and schematic protection systems.

3.3.4 Modeling Dynamic Behavior and Collaboration

The `caretaker`'s proxy behavior in the example of section 2.2.1 is naturally expressed as dynamic behavior: it invokes `carol` and emits (`sendTo()`) to her only what it has collected when it was itself invoked by its clients (`did.receive()`).

Instead of adding more preconditions to the rules, to model how a proxy subject interacts with its target, we will express that proxy behavior with a separate set of behavior rules.

The `PROXY` behavior in figure 2.3 depends on knowledge predicates, that indicate how it got access to the subjects, to decide if it will emit that subject to `carol`. By using monotonic behavior rules, we can model subject behavior as a safe and monotonic approximation, corresponding to the actual code of a software entity.

3.3.5 Modeling n-ary relations

Whereas the permissions in most protection systems are suitably represented as binary relations between subjects, we will need relations of higher arity to model the dynamic behavior of the relied-upon subjects.

The dual-role interaction model of section 3.3.2 reveals this fact, as is illustrated in figure 3.4. An *initiator* `alice` could *emit* `dave` only to `bob` and *emit* `ellen` only to `carol`, thereby depending on her relation with (knowledge about) `dave` and `ellen`. To express this behavior, we will use ternary predicates like :

`alice:may.sendTo(bob,dave)` and

`alice:may.sendTo(ellen,carol)`

Ternary relations, and n-ary relations in general, cannot directly be expressed in a graph-based model like Take-Grant nor in the binary-matrix based HRU systems. Of course, it is always possible to translate n-ary predicates to a set of binary predicates or to represent an n-tuple of subjects as a special kind of subject with n binary relations. Since doing so would only add unnecessary complexity to the modeling process, our new model will simply allow predicates of arbitrary finite arity.

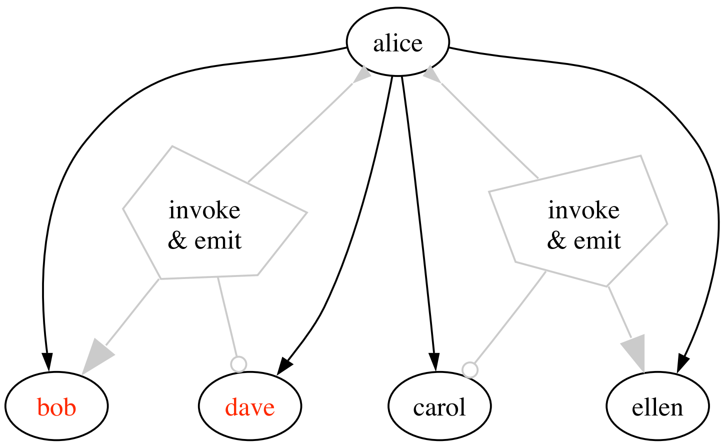


Figure 3.4: Ternary relations are necessary for `alice` to differentiate her behavior.

3.4 Modeling the Permission - Authority Relation

The relation between permissions and authority is very important in a protection system and should be reflected in its theoretical models. In this section we discuss different aspects of that relation that can help us to understand if and why a theoretical model is appropriate to safely and accurately approximate actual protection systems.

3.4.1 The Relation Permission ↔ Access

We use *access* here to indicate the ability to use a permission if one has that permission.

In capability systems and in their Take-Grant based models, permissions (capabilities) imply access. Systems with forgeable designation mechanisms (e.g. pointer arithmetic) also fall into this category, since they cannot prevent access. Access must therefore be assumed in these systems.

In Take-Grant models access also implies the availability of at least one permission and in most actual capability systems this property holds too. Some derived systems have unforgeable designation-only capabilities that can only be used in combination with other capabilities [KGRB02]. Their particularities and practical suitability will not be investigated in this thesis but it will be straightforward to model such protection systems, using the general approach we present in chapters 5 and 6.

In object-capability systems (Section 4.3) the relation permission - access is extremely simple, since there is only one permission: the permission to *invoke*. In such systems access and permission are essentially identical.

A most important implication of this property is: permissions can propagate from subject to subject as easily as access does!

Because permissions must be unforgeable, systems in which access implies permissions should have unforgeable access. Conversely, systems that have unforgeable access can use access to implement (object-) capabilities.

3.4.2 The Relation Permission → Authority

It is common among most protection systems, that the use of permissions guarantees a minimum of authority, e.g. to read the contents of a file.

The actual use of a *read* permission in a given situation may well provide more authority, depending on the contents of the file at the time of its use: it may contain information that is regularly copied from another file so that it also provides *read*-authority to the other file.

Consider what happens if no authority is guaranteed by a permission: would that make the permission useless? In fact, that is exactly what object-capabilities (Section 4.3, [MS03]) do: while the holder of a capability is guaranteed access to the capability's target, that target completely controls the authority it provides and can reduce it to zero if necessary.

Capabilities with guaranteed authority can always be built then by designating target objects that themselves guarantee a minimum level of cooperation. For instance, a conventional *read* capability to a file can be made by building a read-only-wrapper to that file and releasing it as an object capability. Zero guaranteed authority will be useful to implement dynamic policies that require authority-revocation and confinement.

Original Take-Grant systems cannot model capabilities that do not provide a guaranteed minimum of authority. A subject can drop its rights to an object one by one but when the last right is dropped the access is dropped with it.

To respect the dynamic nature of the relation between permission and authority in actual capability systems, we need a model that can take the authority-reducing behavior of the subjects into account.

3.4.3 The Relation Access → Authority

When access implies permission and all permissions guarantee a minimum authority then access implies authority. Take-Grant systems have this property but most practical capability systems do not.

3.4.4 The Relation Permission → Delegation

Delegation of permissions can be controlled by its own specific permissions. Take-Grant systems have *take*-permissions and *grant*-permissions to control two complementary delegation modes: the first for delegation from the responder to the initiator, the second in the opposite direction.

Like the other permissions in Take-Grant systems, *take* and *grant* permissions imply the authority to delegate.

Interestingly, object-capabilities have no separate permission to control delegation: like all authority, the authority to delegate can only be exerted via collaboration between the holder of a capability (the initiator) and the target (the responder). The effect of a delegation attempt is always controlled by the emitter, even if the emitter is not the initiator. Only if the emitter cooperates (either as initiator or as responder) the delegation is established.

The fact that object capabilities have no specific permission for delegation should not lead to the conjecture that object capabilities allow delegation without any permissions. On the contrary, the invoker must have the capability to invoke the responder and the emitter must have the capability to invoke the entity that is being delegated.

The latter guarantees the principle of attenuation: permissions can only be delegated by their rightful holders!

Remark As a consequence of the perception that,in unmodified capability systems *"the right to exercise access carries with it the right to grant access"* [Boe84], Boebert concludes that capabilities cannot be used to guarantee the *-property (*star-property*) of multi-level-security policies.

From the reasoning in his proof we infer that Boebert assumed that *write*-capabilities not only guarantee the authority to write to the target but also the authority to delegate any other capability the holder has to the same target. In a similar way his proof depends on the assumption that *read*-capabilities not only guarantee the authority to read from the target but also the authority to *take* any other capability the target holds.

This conjecture was refuted in [MS03]. Once we have developed a formal system to reason about the propagation of authority in the presence of subjects with restricted, relied-upon behavior, we will formally express and analyze this problem in section 8.4.2.

Chapter 4

Capabilities

Since capabilities were conceived [DH65] there has been a line of research that investigates how holder-controlled capability propagation can be used as the main ingredient to implement secure systems [AP67, Har85, Ree96, SSF99, MSC$^+$01, SDN$^+$04]. These systems include both operating systems and programming language systems.

In this chapter we will first give an overview of the main concepts and principles of such systems, and then concentrate on the safety aspect of this endeavor: how can arbitrary safety properties be enforced by systems that depend only on holder-managed capability propagation?

4.1 The Original Concept of Capabilities

As computers became more powerful in the sixties, and able to run several programs and serve different users concurrently, an architectural layout was proposed by Dennis and Van Horn [DH65] (DVH), based on a set of *meta-instructions* that allow processes to run and interact in safe ways. Their proposal was not only concerned with security in the sense of safety but also with the availability of common resources and with enabling interaction between processes, while avoiding unwanted interference between them.

This section presents the main ideas from DVH, not only for historical interest but mainly to derive the simpler object-capability principles from it which are presented in [MS03]. The ideas of DVH are represented in a frame that allows us to clarify the many claims that have been made about its model, both negative [Boe84, KL87, Gon89, WBDF97, HKN05] and positive [MS03, Mil06b], and to investigate (the need for) alternatives or modifications.

This section only deals with the idea of "capabilities" in that paper and does not go into detail on the other aspects of the proposal. The paper itself is worth reading and studying as a whole as it describes a simple but powerful and consistent set of abstractions that address many of the problems that occur in recent software design.

DVH provide a set of consistent concepts and an initial frame for reasoning about capability based security. We present their work in the light of current knowledge. It is a tribute to the insight of Dennis and Van Horn how well the work stands up.

The meta-instructions we will discuss in some depth here are:

- *create segment*, *create directory*, and *create entry*.
  These instructions create new capabilities

- *delete*
  The instruction to delete owned capabilities

- *create sphere*, *grant*, *ungrant*, and *start*
  These are used to execute computations and processes with a limited set of capabilities.

- *enter*
  The instruction used to start up a process, possibly owned by another principal, within its own *sphere of protection*, while explicitly delegating a capability to it.

- *place*, *link*, *acquire*, and *remove*
  The instructions to widely share owned capabilities.

- *receive*, *owner*, and *transmit*
  The instructions that provide refined ways to share owned capabilities.

In particular, we will not explain the other meta-instructions, but simply assume their general functionality is made available:

- *fork*, *quit*, *join*, *lock*, and *unlock*
  The instructions that provide functionality for concurrency and synchronization

- *private*
  To create variables with local scope.

- *execute i/o function*
  To communicate with external devices

- *halt* and *breakpoint*
  To signal exceptional conditions to the superior (calling) process.

- *fetch status*, *set status*, *continue*, *stop*, *examine*, and *ungrant*
  The instructions for different kinds of error handling.

4.1.1 The Supervisor

Supervisor is the term used by DVH to indicate "*the core of basic computer system functions around which all computations performed by the system are constructed*". The supervisor is responsible for :

- The allocation and the scheduling of resources

- Accounting for, and controlling the use of computational resources

- Implementing the meta instructions.

The part of the supervisor that is concerned with assuring safe cooperation among the system's users and processes, by mediating resources according to a desired security policy, is referred to as the *security kernel* in [Ree96], and will be of most interest to us in this chapter.

4.1.2 Principals and their Processes

Principal is the general term for individual users and groups of users who have access to the machine and are responsible for the use of their allocated resources on the machine. Every principal will have at least one capability: its root directory (Section 4.1.8). What is more, the principal will also have *ownership* of this root directory. What this means will be explained in section 4.1.9.

A *process* is the abstract runtime entity that executes the instructions of a certain procedure. It is, at every moment, associated with a "state word": the information that must be loaded into the processor, to start or continue the execution of the process in its current state.

4.1.3 C-lists and Spheres of Protection

The *state word* contains the process's list of capabilities: its *C-list*. Each entry in the C-list is a capability that combines:

- A means to designate some computing object
- The actions that the process is allowed perform on that object.

The actions are related to the sort of computing object that is pointed to, and are classified into different types of capabilities. (Sections 4.1.4 to 4.1.8).

It is possible that more than one process shares the same C-list. That is for instance the case when one process has spawned another process using the *fork* meta-instruction. The complete set of processes that share a C-list is referred to as a *Computation*.

The C-list defines a "*Sphere of Protection*" in which the computation proceeds. There is a one-to-one correspondence between Computations, Spheres of Protection, and C-lists. However, we will see in section 4.1.6 that a process can sometimes call (invoke, start) another process in a different Sphere of Protection.

To use the capabilities in its C-list, a process must apply the appropriate meta-instructions implemented and provided by the supervisor. The process can only use capabilities from its own C-list, simply because it can only reference capabilities via their index number in its own C-list.

There are no meta-instructions that allow a process to make changes to an existing capability, neither to the designation nor to the allowed actions.

Types of Capabilities

4.1.4 Segment Capabilities

DVH assume that the accessible memory is addressable via *words* (the smallest units of accessible memory), that are grouped into ordered lists called *segments*, for the purpose of naming. A word is to be referenced by *word name*: a simple combination of the index number of the segment capability and the word's sequence number in that segment's list.

A segment capability designates such a segment and allows actions from any subset of $\{X, R, W\}$ except $\{W\}$ and $\{X, W\}$. X indicates permission to execute, R permission to read, and W permission to write.

Why these two particular sets are excluded is not clear from the paper. Possibly DVH assumed that there would be no reasonable use for them.

Segment capabilities can be created using the *create segment* meta instruction which takes a valid set of permitted actions as input, allocates a segment (if resource allocation allows it), creates an appropriate segment capability in the computation's C-list, and returns the entry number of the newly created capability.

No particular meta-instructions for *using* segment capabilities are provided, which may indicate that DVH considered them to be either trivial to implement by the supervisor or not the supervisor's responsibility.

In the paper it is implicitly clear from the context that the C-lists themselves, while being part of a state-word and therefore being available as data to the supervisor at some level of abstraction, are not to be kept in memory segments to which the supervisor will create segment capabilities. Otherwise the supervisor would of course not be able to effectively implement the necessary protection mechanisms explained in this section.

4.1.5 Inferior Sphere Capabilities

A process that is initiated with the *fork* instruction executes in the same sphere of protection as its parent process (the one that gave the *fork* instruction). DVH provide a way to avoid this, and start a process with no capabilities at all.[1]

The *create sphere* instruction can be used to create a fresh sphere of protection (empty C-list). It will make the new (inferior) sphere available as an inferior sphere capability in the C-list of the parent (superior) process.

Once the parent process has access to an inferior sphere capability, it can use the *grant* instruction to add a copy of one of the capabilities in its own C-list to the inferior C-list designated by that inferior sphere capability. The parent process can choose for the copy to be partial, in the sense that set actions in the copied capability can be a subset of the one in the original capability. If the capability is owned by the parent process, the parent process can choose to grant the ownership too. Section 4.1.9 will explain about ownership of capabilities.

When the parent process decides it has granted enough of its own capabilities in that way, it can use the instruction *start* to initiate a new process in the inferior sphere of protection. Notice that this means that the "superior" and "inferior" processes will be in different computations (Section 4.1.2).

DVH illustrate the utility of inferior spheres when calling an untested procedure under construction during debugging, but of course it applies in general to all situations in which POLA is to be applied.

4.1.6 Entry Capabilities

DVH's motivation for introducing entry capabilities is their aim to allow data abstractions and devices to be used by several processes while ensuring that these processes remain protected from each other.

A similar concern is described as "defensive consistency" in [Mil06b], or as its stronger variant "defensive correctness" when the commonly used abstraction can also protect itself from a denial of service attack. If two computations A and B both make use of a routine S and are not otherwise dependent, it must be possible to program S so that a malfunction of A's processes cannot cause incorrect execution of B's procedures.

[1]This statement will be reviewed in section 4.2.

Part of the concern is that S's private data should not be modifiable by A or B. This is the concern that is referred to as the need for encapsulation in software engineering.[2] But S may also be designed to modify data that has to be available by its client's process, which means that there should be a way to for its client to add a capability to S's C-list.

To solve this problem DVH provide the atomic meta-instruction *enter*, in which the following are combined:

- Making the necessary change to S's C-list
- Relinquishing control to S at a proper entry point of S's operation
- Changing the C-list-association of the running process : from the C-list of S's client to S's own, modified, C-list

Entry capabilities can only be created with the *create entry* instruction, which takes a word name (Section 4.1.4) as input and a positive integer to indicate where the legal entries are situated. The instruction creates an entry capability, adds it to the C-list of the current process, and returns its index number in the C-list.

Only the owner of the segment capability referenced by the word name is allowed to create entry capabilities. The concept of ownership will be explained in section 4.1.9.

The *enter* instruction is provided for using an entry capability. It takes as input:

- The index number of the entry capability in the calling process's C-list
- A number indicating what entry point of the called procedure is to be invoked
- The index number of the capability in the calling process's C-list the caller wants to make available (delegate) to the called procedure to enable it to perform its task.

The instruction performs the three combined tasks above. On top of the delegated capability it also adds a *suspended process capability* to the called process's C-list to enable it to relinquish control back to the caller after the computation.

To return from the call the called process uses the *continue* instruction which it will give the index of its suspended process capability as input. That instruction will return control to the caller after having deleted the suspended process capability from the C-list. Notice that *continue* does not delete the delegated capability from the computation's C-list.

4.1.7 Receive Capabilities

Receive capabilities are place-holder-capabilities, created by the calling process to be filled in by the called process. They are created by the *receive* instruction that takes no input arguments, creates a receive capability, puts it in the executing process's C-list, and returns its index number in that C-list. The calling process is supposed to pass this capability to the called process using the *entry* instruction (see above).

A receive capability can only be used by a process that has a suspended-process-capability to the calling process that created the receive capability. Before using the

[2]It is commonly assumed that Dave Parnas introduced information hiding as a software design goal in [Par72]. Apparently, DVH [DH65] already identified a good motivation for the concept in the 1960's

continue instruction the called process can use the *transmit* instruction to fill in the place-holder represented by the caller's receive-capability.

The *transmit* instruction takes three input arguments:

- The index in the current C-list, of the suspended-process-capability pointing to the caller of the current process.
- The index in the current C-list, of the receive capability that was *entered* by the caller of the current process.
- The index in the current C-list, of the capability that will be transmitted to the caller of the current process.

If the receive capability and the suspended-process-capability correspond, the receive capability in the C-list of the process that called the process that uses the *transmit* instruction is replaced by the transmitted capability.

Note that receive capabilities can be passed on to other processes again via *enter* capabilities.

Remark

DVH introduce receive capabilities and the *transmit* instruction at the very end of the paper, when they discuss a mechanism for sharing capabilities between principals based on the identification of the principal that requests a capability. Contrary to the capabilities that can be transferred using the *entry* instruction on an entry capability, the capabilities that can be transferred using the *transmit* instruction on a receive capability must be *owned* (*O*) in the C-list of the process that uses the instruction.

The difference between the transfer of owned versus non owned capabilities will be discussed in section 4.1.10.

4.1.8 Directory Capabilities

Directory capabilities are a means of grouping and naming capabilities in a hierarchical way. The can be create with the *create directory* instruction that takes no input, creates a directory capability, adds it to the C-list, and returns the index in the C-list.

Adding a capability to a directory is done with the *place* instruction, that takes as input:

- The index of the directory capability in the C-list
- The index of the added capability in the C-list
- The name to be given to the added capability's entry in the directory
- An indication, either P for private or F for free, that indicates the intention to share the capability that was placed in the directory.

To use the *place* instruction both the directory capability and the added capability must be *owned*.

Removing a capability from a directory is done with the *remove* instruction that takes as input the index of the directory capability in the C-list and the name of the entry in that directory that has to be removed from the directory. To use the *place* instruction the directory capability must be *owned*.

4.1.9 Ownership of Capabilities

An entry *i* in a C-list can be marked (*O*) for *owned*. This happens exactly in the following situations:

1. When *i* indicates the root directory capability of the principal on whose behalf the supervisor initiated the current computation. Every principal has exactly one root directory and owns this directory.

2. When *i* indicates a capability that was created with the *create directory* or *create segment* instruction.

3. An entry *j* to a segment capability is marked (*O)* in the C-list and *i* indicates a capability that was created with the *create entry* instruction using *j* as the segment input for that instruction.

4.1.10 Propagation of Capabilities

We can distinguish between two ways of sharing capabilities in DVH. First there is the *enter* instruction that allows a process to share (delegate) all its capabilities regardless of ownership but only with a process or computation it has an entry capability to. The other mechanism proposed in DVH uses directory capabilities to share owned capabilities only.

Sharing of Owned Capabilities between Principals

DVH use directory capabilities not just for ease of naming and grouping but also to enable the sharing of owned capabilities between processes and principals. The intention is to allow principals to make their data and procedures available to other principals. There are two mechanisms to do so:

Sharing with all principals A process can make a capability available to all principals if that capability is marked (*O*) in its C-list. It does so by adding an entry for the capability into its root directory and by marking that entry (*F*) for free. Adding an entry into a directory is done using the *place* instruction which takes four input arguments:

- The index of the a capability that will be added (must be marked (*O*) in the C-list).
- A name to indicate (retrieve) that capability from the directory.
- The index of the directory capability itself (must be marked (*O*) in the C-list).
- An element from $\{P,F\}$ to mark that entry in the directory. P stands for *private*, F stands for *free*. Only *free* entries are shared with other principals.

Sharing with specific principals To allow principals to make their owned capabilities available to selected principals only, DVH suggest the instruction *owner* that takes as input the index of a receive capability in the current C-list and returns the name of the principal that created the receive capability. The instruction makes no changes to the current C-list.

To share an owned capability with a specific principal, the owner will write a purpose build procedure into an *owned* segment and place and X-capability (execution) to that segment marked as (F) (*free*) in its root directory, available for every other principal via the instruction *link* and *acquire*. Every principal that wants to use the actual capability can invoke that procedure with a receive-capability but the owner's procedure can use the *owner* instruction on the receive-capability to check on which principal's behalf the requesting process is running and decide whether to *transmit* (return) the requested (owned) capability.

Sharing of non-Owned Capabilities via Entry Capabilities

Once a process has in its C-list an entry capability to a procedure in a segment owned by another principal, it can use the *enter* instruction to delegate the capabilities in their C-list to the process that was started by calling that procedure. The *enter* instruction takes only one capability. Contrary to the intuition created by a first read of DVH, that limitation cannot simply be circumvented by grouping several capabilities.

Here we find an indication that DVH *may* have designed directories to serve two purposes that should better have been separated: the grouping of capabilities into one capability and the sharing of owned capabilities. We are aware that we cannot make strong claims about this, as we cannot infer anything concrete about DVH's actual *intentions* from the paper.

To understand why it may have been better to separate these aspects, it suffices to consider the difference between propagating a segment capability using *enter* or instead using the same instruction to propagate a directory capability that contains exactly that segment capability. In the former case the segment capability does not have to be owned while in the latter case it does. To be useful in the latter case the segment capability also has to be marked free (F) in the directory capability. It does not seem very useful to pass a private capability using *enter* this way.

4.2 Interpretation and Discussion

There is a tension between DVH's goal to protect processes from each other's malfunction or bad intentions, and their goal to facilitate the sharing of capabilities between principals. Concrete: the *link* and *acquire* instructions can be used inside an inferior sphere of protection (Section 4.1.5) to *acquire* the same capabilities that the superior sphere of protection can acquire.

Since capabilities can only be named in directories, a possible interpretation of the supervisor's job is: he will put all capabilities he wants to enable the principal to use into the principal's root directory instead of directly into the C-list of the process he starts up on behalf of that principal. In fact this means that a process running inside an inferior sphere of protection can always undo all its limitations by using the *acquire* instruction itself. Thus it is impossible for a process to *exactly* control (POLA) the set of capabilities that will be available for its subprocess.

This situation is called *ambient authority* : processes can have authority, in this case in the form of capabilities, that was neither explicitly given to them by their parent process nor acquired as a result of using the authority explicitly given to them by their parent process. This was brought to our attention by Charles Landau in a mail [Lan06] to the cap-talk mailing list [Cap].

There are two ways to solve this flaw:

1. Make the *acquire* instruction unavailable inside all inferior spheres of protection.
2. Allow the programmer to make the *acquire* instruction unavailable inside a specified inferior sphere of protection, upon *creation* or *start* of the sphere.

It is possible to divide the subsequent research efforts on capabilities in two camps. There is the camp of "believers" who concentrate on the essentials in the DVH model, without the mechanisms for ownership-based sharing, and there is the other camp that assumes DVH capabilities are too permissive for practical use, and who try to fix that problem by adding modifications to the model itself.

4.2.1 Fixing ambient authority

We cannot claim with certainty that it was indeed *not* DVH's intention to allow ambient authority, but we do claim that their model makes the most sense when interpreted as *not intentionally* providing ambient authority, because the inferior spheres were introduced exactly to protect a process when running code it did not want to rely upon. This is obvious from the debugging motivation given in section 4.1.5.

By avoiding the ambient authority problem inside inferior spheres of protection only, all proposed mechanisms to share capabilities between principals can be kept in the computation that runs in the principal's superior sphere of protection. This computation corresponds the principal's *shell* process, started by the supervisor on behalf of the principal. By default all processes started by this shell process should be ran in their own inferior sphere of protection, to allow the principal to run untrusted code.

The capabilities that can be acquired by the main user process (shell) are then to be considered as a fixed and static package of authority, available by "initial conditions" (will be explained in section 4.3.2), as opposed to ambient authority. Section 4.3 shows a simplified model that is directly derived from DVH, that takes initial conditions into account, but has no ambient authority.

4.3 Object Capabilities

We present here the object capability model used in [MS03, Mil06b], and relate it to the original DVH model. Object capabilities are a purified and simplified version of capabilites. They will form the basis for the extensible formal model of chapter 5. The main advantages of the object capability model are its simplicity and the complete avoidance of ambient authority.

The object capability model abstracts from the actual mechanisms for storing, retrieving, and sharing capabilities, and only considers the abstract rules that govern the use and propagation of capabilities. In that model we refer to computational entities like processes, computations, objects, and devices, as subjects. If the level of granularity of the subjects is coarse enough, we can use them to represent principals, more precisely, the set of processes and computations running on behalf of a principal.

We no longer consider the concept of ownership though we will see that some rules only apply in situations that correspond to ownership in DVH.

4.3.1 One Type of Capabilities

All capabilities designate entities of the same type: subjects. All capabilities combine a single right with this designation: *access*. This implies that there is only one type of capability: the one that provides the permission to *access* a subject.

The *access* permission indicates that the holder of the capability is allowed (and able) to use the target subject in any way possible. What exactly is possible will depend on the target subject.

Object capabilities work like entry-capabilities in DVH. If a called process has a segment-read-capability it can use that capability on behalf of its caller and copy the contents using a segment-write-capability that was delegated by the caller in the same *enter* call. In that case the entry capability would provide segment-read-authority to its holder.

This is a form of collaborative authority, because it is realized by the behavior of the holder and the target of the capability. With object capabilities every authority stems from collaboration.

4.3.2 Capabilities Available by Initial Conditions

We assume an initial configuration exists that safely approximates the capabilities that are available to the subjects at the start of the analysis.

Whatever mechanism was responsible for creating these initial conditions (e.g. the supervisor), we assume that only the mechanisms described in the following sections (Sections 4.3.3 and 4.3.4) are active from then on.

4.3.3 Acquiring Capabilities by Parenthood and Endowment

Parenthood

Subjects can create new subjects and, by the act of creation, get a capability designating the created subject. Acquiring a capability by the act of creating a new (child) subject is referred to as *parenthood*.

In capability based programming, creation corresponds directly to the instantiation of an object or a procedure by calling a constructor method or by evaluating an expression.

A subject comes to life with no default authority, more precisely it does not by default inherit the capabilities of the parent subject that created it. This mechanism is not directly supported in DVH.

An entry capability representing the child could be created inside a newly created inferior sphere of protection and communicated back to the superior sphere's C-list (parent). Alternatively, an entry capability representing the child could be created in the creator's sphere of protection, in such a way that it will immediately proceed its own execution in a new sphere of protection.

Endowment

The parent subject can choose to share some of its capabilities with the child subject it creates.

In memory safe programming languages, endowment can correspond to:

- The instantiation of an object or a procedure via a constructor method with an input argument designating the endowed capability.
- The evaluation of a lambda expression in a context with free variables containing the endowed capability.

In DVH, endowment corresponds to *granting* the endowed capabilities to the inferior sphere of protection mentioned above.

4.3.4 Acquiring Capabilities by Interaction

The main credo of object capabilities is : "*Only connectivity begets connectivity*". Subjects can only acquire extra pre-existing capabilities by interacting with other subjects.

Every propagation involves two subjects, each of them playing a role as either *emitter* or *collector* of the propagated capability, and each of them also playing a role as either *initiator* or *responder* in the interaction. These two pairs of roles are independent but all roles must be filled: either the emitter or the collector can be the initiator but only if the other one is the responder.

The conditions for successful propagation of an object capability designating subject $carol$ from subject $alice$ to subject bob are:

- either:
  - $alice$ has access to $carol$, and
  - $alice$ has access to bob.

  In that case $alice$ plays the role of initiator-emitter and bob plays the role of responder-collector.

- or:
  - $alice$ has access to $carol$, and
  - bob has access to $alice$.

  In that case bob plays the role of initiator-collector and $alice$ plays the role of responder-emitter.

Notice that these propagation rules are very similar to the take and grant rules in figure 3.1 of section 3.2.2. This similarity deserves an explanation. It may seem that the object capability model is a simplification of the Take-Grant model, without the *take* and *grant* capabilities. However, this simplification is only part of the story.

Remember that Take-Grant models had two types of subjects: active and passive ones, and that passive subjects (also called "objects" there) could prevent some kinds of propagation. The object capability model transforms the static, subject-type-based propagation rules in Take-Grant systems into dynamic, behavior based propagation by imposing the following extra dependencies on subject behavior:

- The emitter *chooses* which subject will be propagated, if any.
- The initiator *chooses* which subject it will interact with, if any.

4.3.5 The Authority attainable by using Capabilities

The subject-type-based propagation of authority in Take-Grant systems is refined in the object capability model. The rules that govern propagation explicitly depend on subject behavior:

- The subject holding a capability *chooses* in which circumstances and to what effect it will use the capability.

- The subject designated by a capability *chooses* in which circumstances it will cooperate in realizing what authority.

This allows for a tractable, behavior-based safety analysis whose accuracy depends on the level of detail used to model the behavior of the entities.

Instead of a write-capability designating a file, we can use an object capability to a write stream that has an object capability to a file but can be relied-upon to only ever use that capability to write to the file and never to read from it. This gives us more expressive power.

We can refine the write-stream instance's behavior to write only in certain cases, for instance when its caller can provide a special token-only capability as a proof of trustworthiness. The use of a token capability in this example is similar to the use proof-of-knowledge-of-a-common-secret (password, symmetric key) in encryption protocols, to authenticate authorized users.

When refining the write-stream instance's behavior is not an option, wrapping it into a proxy can be an alternative. Authority revocation is a powerful example of how "authority reducing wrappers" can be used. The wrapper (proxy) can decide to completely stop cooperating and thus effectively revoke authority from its holders. This is one of the examples that will be analyzed in chapter 8.

Propagation by Interaction in DVH

The *enter* instruction allows the initiator to delegate a capability to the process designated by an enter capability in the initiator's C-list. If the delegated capability is an entry-capability, then the initiator is the emitter and the responder is the collector. If the delegated capability is a receive-capability, then the initiator is the collector and the responder is the emitter of an entry-capability using the *transmit* instruction on the delegated receive-capability..

The code being executed in the initiator's process decides what entry capability to use, and what entry capability to delegate. The code being executed in the emitter's process decides what entry capability to propagate.

The code being executed in a process decides what entry capability to use and to what purpose. The code being executed in the called process decides what will happen when it is called.

4.4 DVH as Object Capabilities

In section 4.3 we have systematically expressed object-capability rules and mechanisms in DVH. This shows that object capabilities can be expressed in the original capability model. To prove that the original capability model has the same nice properties as object-capabilities, we need to show the converse: how DVH can be expressed using object-capabilities only.

The DVH paper was only a conceptual paper that introduced the idea of abstractions for access control, based on capabilities. It is therefore not surprising that we have to restrict or refine some mechanisms in DVH to precisely express their model using object capabilities.

We will interpret DVH's mechanism for sharing capabilities between principals via directories in its restricted form, as a supervisor-controlled way to provide initial conditions, instead of as ambient authority (Section 4.2.1). We will also remove the concept of ownership and all related restrictions.

The remaining tasks of the supervisor can be expressed directly in the object capability model.

- The segment-capabilities can be implemented as supervisor-provided object capacities that are the only ones with direct access to the segment and are relied-upon to restrict their *use* of the segment corresponding the appropriate subset of $\{R, W, X\}$ permissions.
- The *enter* and *transmit* instructions can be implemented directly with object capabilities with appropriately confined behavior.

4.5 Restricting the use and the propagation of capabilities

It is strongly suggested in many security papers [KL87, Gon89, WBDF97, HKN05] that capabilities need a "modification" to enable them to impose confinement in a straight forward way. Some propose to add holder-identification mechanisms to capabilities in order to restrict the use of capabilities to rightful holders. Others add specific permissions to control the propagation of capabilities. Most of these papers refer to Boebert [Boe84], for a proof that the $*$-property (Section 8.4.1) cannot be imposed in a straight forward way in capability systems, without such modifications.

This is allegedly caused by the "fact" that "the right to exercise access carries with it the right to propagate access", which is interpreted by Boebert as : "the permission to exercise access carries with it the authority to propagate access". We were not able to find anything in the DVH paper that supports this proposition.

We assume that a confusion between permissions and authority may have lead to this conjecture.

Section 8.3 will demonstrate the ability of capabilities to enforce several forms of confinement, including the $*$-property. It will also show why this ability cannot be demonstrated in models that do not, or to an inappropriate level of detail, take behavior restrictions into account.

4.6 Capabilities compared to Access Control Lists

In this section we will clarify the difference between capabilities as explained in this chapter and the approach based on a dedicated reference monitor to control access permissions by checking an access control list (ACL) at runtime. The latter is more common in current operating systems.

A first and seemingly superfluous difference is the organization of the permissions: capabilities keep the permissions with the subject that holds the permission, whereas most implementations of ACLs store the permissions with the target ("object") of the permission. If this would only be a difference in implementation, both approaches would be conceptually equivalent.

Capabilities combine designation with access rights whereas ACLs keep these concepts separated. The ACL approach therefore seems to have the advantage of promoting the separation of concerns between functionality (what the programmer wants his program to do) and security (what the system administrator wants the program *not* to do).

Unfortunately, it is not always appropriate or even possible to separate the concerns of functionality and security in this way: sometimes we may want an entity to have permission to do something for a certain purpose but not for another purpose. This purpose cannot simply be derived from runtime information that is available to the reference monitor, even when that information is extensive [WBDF97]. The intention of the entity is only known to the programmer of the entity who should be able to build the entity so that it cannot suffer a confused deputy attack.

The confused deputy attack was first described in [Har88] and is also explained in [Spi]. It describes client entities abusing a service entity's authority by designating an entity (e.g. a file) to which the service entity has access rights but the client entity has not. The service entity then assumes that its permission to access that file was delegated by its client and uses it for the client's purposes.

This confusion can be avoided if access permission and designation are combined because in that case the client entity would not have been able to designate the file in question. The service entity (deputy) is sure to use its own permissions by using its own designations (capabilities), and to use the permissions delegated by its client, by using the designations (capabilities) provided by its client. Section 8.1 will explore confused deputies concept in depth.

Other important advantages of the capability approach are:

Simplicity The runtime checking of permissions is very straightforward since it suffices to make capabilities unforgeable and to allow no other form of designation to provide access. In memory safe programming languages, since references to entities cannot be forged, the runtime system plays the role of a reference monitor already.

Complete Mediation No access can be forged and escape mediation by the capability mechanism.

Extreme POLA The principle of least authority can be applied up to the level of the finest grained entities: procedures and objects.

Because of the dynamic nature of authority, controlling the (dynamic) behavior of relied-upon subjects is the most appropriate way to exert authority control. It allows us to apply POLA, not only at the level of individual subjects, but at the level of their intentions. In contrast, most ACL based policies only control users and resources and allow every program to run with all the authority of the user that started it.

Small TCB The TCB (Trusted Computing Base) indicates the set of components of a system that are completely relied upon by all other components for their safety, and have to be implemented in a trustworthy fashion. Applying POLA consequently at all levels of granularity is a sure way to shrink the TCB.

Many advantages of capabilities are explained in detail in [MTS05].

Object capabilities have the disadvantage that they cannot confine the authority flowing between two untrusted subjects, once one of them has a capability designating the other. That is because, in that situation, no restrictions in the behavior of any of the subjects can be relied upon. In capability based systems the aim will always be to prevent this situation from occurring by keeping at least one relied-upon subject interpositioned between both untrusted subjects to restrict the authority flow between them. Section 8.3 gives some examples of how this goal can be achieved.

The fact that object capabilities have no specific permission to control the propagation of capabilities or data may seem to be a reason for concern: how can we ever make sure that two entities are kept from having a certain influence on each other, if we cannot express that type of control as a directly enforceable permission? Does that not complicate things, since we will have to analyze the boundaries of authority propagation up front?

The relation between permission and authority is intricate and dynamic, and only permissions can be directly controlled. Even if we have fine-grained permissions, an analysis of the authority that can become available will be necessary.

This analysis can reveal where in the initial access graph relied-upon subjects need to be inter-positioned and what the behavior restrictions of the relied-upon subjects should be.

Part II

Main Contributions

Chapter 5

Knowledge Behavior Models

In this chapter we present "Knowledge Behavior Models" (KBMs), a new and practical formal model, in which the safety problems that are of interest to software engineers can be expressed and analyzed.

KBMs are used to express how some relied-upon components in existing software may possibly propagate authority and then to predict how authority cannot propagate in that software. KBMs principal advantage is in discovering how the relied-upon restrictions in the software components will affect the predictions about the complete software. The result is a set of realistic design requirements for the components that have to be developed (or need to be adapted) that will guarantee that no illegal authority can be attained in the software.

This chapter is largely self contained. However, an understanding of the concepts introduced is Section 1.3 is required. A previous reading of the chapters 1 and 2 is recommended to understand the contribution in its context.

Some of the new concepts introduced here will be revisited in the more practically oriented chapter 7, using the description language SCOLL that will be introduced in chapter 6. We refer to chapter 8 for a list of well explained, useful, and illustrated examples of how KBMs, expressed in SCOLL, are used for safety analysis.

The Structure of this Chapter

The first section of this chapter provides a motivation for KBMs, illustrated with programming code examples. Section 5.2 will then give a concise overview of the concepts and mechanisms we use in our approach

Section 5.3 will introduce the basic components that constitute a KBM. Sections 5.4 will further clarify the role and the use of these components in a simple capability based example.

Section 5.5 shows a refined version of the example to illustrate the technique of refinement. The process of progressive refinement of a KBM is a crucial part of our approach. Section 5.6 will present a general account of refinement.

Section 5.7 will then give the formal definitions of KBMs and of the safety problems that can be expressed using KBMs. It will also provide a formal definition of aggregation and a formal proof that aggregation of safe models always results in a safe model.

5.1 Motivation

KBMs were developed as a synthesis of the formal models described in earlier chapters, combining their desirable properties and avoiding the drawbacks that made them impractical for expressing and analyzing the safety concerns of software engineers.

We want our formal model to serve a practical purpose: to help designers and developers of secure software to understand the actual requirements for the software entities they rely upon to enforce their safety goals, given a context (pattern) in which these entities will play their role. Relying on programmed restrictions in the software improves the expressive power of authority analysis and the effectiveness of authority control in comparison to an approach that restricts its attention to the availability and the propagation of permissions.

In "Robust Composition" [Mil06b] Miller compares the possible approaches to safety analysis and concludes that our approach is preferred, because it allows the tractable calculation of a non-trivial upper bound on eventual (reachable) authority that becomes more accurate when behavior is modeled in greater detail. We will consequently pay more attention to expressing behavior than to refining permissions.

KBMs are particularly suitable to model software that is controlled by a capability based protection system. Capability based protection systems are relevant for software engineers, because the implementation of such a system can be embedded in the language runtime, as is the case for capability secure languages like E [MSC^{+}01], Emily [SM06], and Oz-E [SV05].

Even so, the models remain general enough to express many protection systems. Alternative forms of permission control (via reference monitoring) can impose further (or other) permission-based restrictions. KBMs will consequently allow us to make formal comparisons between alternative approaches for protection systems.

To illustrate our approach, we will provide code examples in Oz-E [SV05], the capability-secured subset of Oz [Moz03, VH04] or in Emily [SM06], a capability-secured subset of OCaml[CMP00]. Both are multi paradigm languages, suitable to illustrate how KBM subjects can model different kinds of software entities (objects, modules, functions, procedures, etc.) and different mechanisms for authority propagation (e.g. invocation and message passing).

Oz has preemptive threads and logic variables, specific language constructs that allow us to illustrate how differences in programming languages can influence KBMs. Emily's restrictions to OCaml guarantee that no ambient authority is attainable in the software and thus allow us to express examples that only depend on a pure capability based protection system.

5.1.1 A Preliminary Example

A simple code example in Oz and Emily will provide additional intuition about our goal and motivation.

The top part of figure 5.1 shows the Oz code for a procedure `AliceProc`, an object `AliceObj`, and two unspecified entities `Bob` and `Carol`. Below it, an equivalent situation in Emily is shown. We will refer to the entities in abstract terms as the *subjects* `alice`, `bob`, and `carol`.

Modeling the Initial Access Graph

We will assume that the runtime environment for the software does not provide ambient authority. That means that, upon loading the modules that will create `bob` and `carol`, these modules get no access to other entities. In Emily, all ambient authority has been removed from the OCaml environment, but in Oz, the programmer will have to implement his own loader (Module Manager) to avoid ambient authority.

The loaded modules need no ambient authority to create `bob` and `carol`, but they can (only) convey to their created entities the authority that they have themselves. Therefore it is possible that `bob` has access to `bob` and `alice` has access to `alice`.

Furthermore, `bob` and `alice` can each have access to the module that created them, and to other entities that may be created by the same module. We ignore these complications for now, and assure the reader that they will be dealt with soon, in an elegant way.

In the KBM, we will model all *possible permissions* between entities as *actual permissions* between subjects, because we want to use the model to calculate an *upper bound* to the authority that is reachable in the software.

From the code that defines `alice` (as a procedure or an object), we can infer that, immediately after its execution (when `alice` is defined), she only has access to `bob`, not to `carol` or to herself.

The access graph of figure 5.1 depicts the initial access permissions of the subjects. Note that KBMs will allow us to describe the initial state of the subjects in greater detail than is suggested in this figure. It can include all kinds of initial conditions to model the initial state of the entities more accurately.

Behavior

Looking at the code for the entities corresponding to `bob` and `carol`, we see that they are created from external modules. We are unable (or unwilling) to express any specific knowledge about these modules in the model. For the purpose of authority analysis, we must therefore assume that `bob` and `carol` will do everything they possibly can, to break our security requirements.

Unknown (untrusted) entities, like `bob` and `carol`, will be modeled as abstract subjects with *unrestricted behavior*. In graphs, we will depict the name of unknown subjects in red, and the name of relied-upon subjects in black (see the access graph of figure 5.1).

We will model the restrictions of relied-upon entities as the behavior of relied-upon subjects. For that, we need to perform only a *local inspection of the code* of the entities that are modeled. Conversely, to program entities that satisfy the behavior of modeled subjects, we only have to inspect the subject's behavior. This is crucial, because we want to make our approach useful for programmers, during the design and the implementation phase of their software.

For instance, by looking only at `alice`'s code, in either language, and in either implementation (as object or as procedure), a programmer can derive the following restrictions, he can rely upon:

1. `alice` will only accept input arguments, and not return anything to her clients (invokers, senders of messages). In the Oz example, since Oz has logical variables that can be passed on before they are bound to a value, the programmer

```
declare
   Bob = {{Link ["bobModule.ozf"]}.1.makeBob}
   Carol = {{Link ["carolModule.ozf"]}.1.makeCarol}

   AliceProc =  proc{$ Input}
                       {Wait Input}
                       {Bob Input}
                end

   AliceObj = {New class $
                      meth doSomethingWith(Input)
                         {Wait Input}
                         {Bob doSomethingElseWith(Input)}
                      end
                      meth init() skip end
                   end
               init()}
```

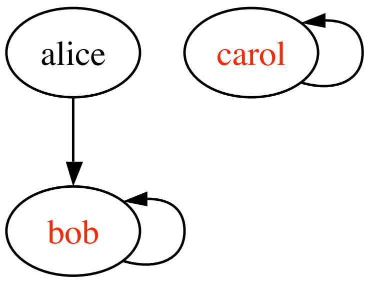
`bob`, `carol` and the relied-upon subject `alice` (x 2) in Oz

```
let bob = BobModule.makeBob();
let carol = CarolModule.makeCarol();

let aliceProc input = ignore (bob input);

let alice = object
            method doSomethingWith input =
            ignore (bob doSomethingElseWith input);
```

`bob`, `carol` and the relied-upon subject `alice` (x 2) in Emily



The initial access graph

Figure 5.1: Preliminary Example

made sure that the variable is bound *before* it is passed on in `alice`'s code, by using the `Wait` procedure. That procedure will block the execution of the thread until `Input` is bound.

2. `alice` will only use the access permissions that designate an entity referred to as `Bob`.

3. `alice` will only use her permission(s) to supply (relay) the input arguments she accepted. No return values are expected from the invocation (or from the message send). Even if `bob` would return a value, `alice` would completely ignore it. In the Oz example, the `Wait` procedure ensures that `bob` cannot bind the variable `Input`, because logical variables can only be bound once. If `bob` could bind a value to the logical variable `Input`, that would be equivalent to returning a value directly to `alice`'s client.

In the next sections, we describe how the entities are modeled in a KBM, how their behavior is expressed in the model, and how the protection mechanism (that controls the use and the propagation of the permissions) will also be modeled. When all these aspects of the software are modeled as a KBM, the programmer's safety concerns will be expressed.

Everything will then be ready to:

- Calculate an upper bound to the propagation of authority.
- Check if the safety concerns are guaranteed.
- Calculate the minimally restrictive (maximally permissive) adaptations to the behavior of the relied-upon subjects that are sufficient to guarantee the safety concerns.

5.2 Approach

In this section we give an overview of the approach we use to model software in KBMs. For conciseness, this section contains no examples. All elements of a KBM will be explained in greater detail in the next section, with examples referring to figure 5.1.

Preliminary Remark: Interaction From now on, all ways in which authority can possibly propagate in the software will be called *interaction*, regardless of the actual mechanism that is used and regardless of the number of entities that are involved in the propagation of authority.

KBMs will model how software entities can interact, given the restrictions imposed by a runtime protection system and the relied-upon restrictions that were (or will be) programmed into the entities. They will be used for analyzing what authority can possibly become available as the effect of these possible interactions.

5.2.1 Safe and Tractable Approximations

Because KBMs will be used to prove safety properties in the modeled software (prove that authority is not reachable), we must make sure the KBM is a safe and tractable approximation of the software: all authority that is reachable in the software must be reached in the KBM in a time polynomial in terms of the size of the KBM.

Therefore, we use the following approach, when modeling software as KBMs:

1. Choose a simple model for the interaction between the entities in the software that corresponds naturally to the interaction model of the programming language. In most examples we will restrict ourselves to a send-and-return interaction model. For languages that use continuations, a simpler "invoke" model, will suffice. The latter is more primitive. A remark in section 5.3.2 will show how the send-and-return model can be expressed using continuations.

   The interaction model should cover all ways to convey authority between the entities. For instance, if the language has shared state (e.g. cells with mutable content), updating and/or reading the state should be modeled as a form of interaction, in which the cell is involved in a collaborative but passive way.

2. Model all software entities as KBM subjects, including the ones that are created at any stage, in any possible execution of the software. It will be possible to model multiple software entities as a single subject.

3. Model the preconditions for, and the postconditions of successful interactions in the interaction model. Model all conditions as predicates over the subjects (labeled relations). At least, all permissions used by the protection systems should be modeled as predicates.

   Beside permissions, three more types of predicates should be considered, corresponding to three types of authority:

   (a) The positive influence entities have on interactions.

   (b) The positive influence successful interactions have on the behavior of entities.

   (c) The positive influence entities exert directly on themselves, outside the interaction model.

   Predicates of type (a) will be called *behavior predicates*, the latter two types: *knowledge predicates*.

   less

4. Model the rules for successful interaction imposed by the protection system as *system rules*. Every system rule is a set of preconditions and a set of postconditions. The pre- and postconditions consist of the predicates defined earlier. Preconditions of a system rule can contain only permission predicates and behavior predicates. Postconditions of a system rule can contain only permission predicates and knowledge predicates.

   The semantics of the system rules :

   - If all the permissions among the preconditions in the rule are satisfied, the interaction is allowed.

- If all preconditions in the rule are satisfied, the interaction is executed.
- Executing the interaction will cause all postconditions to be satisfied.

5. Model, from what is known about the software, a safe approximation of the dynamic influence of every software entity on its interactions as a set of behavior rules for the subject that models the entity. Behavior rules are similar in structure and semantics to system rules, but their preconditions contain only knowledge predicates and their postconditions cannot contain knowledge predicates of type (b).

   The semantics of a behavior rule for a given subject :

   - If all preconditions in the rule are satisfied, the behavior rule of the subject is executed.
   - Executing the behavior rule of a subject will cause all postconditions to be satisfied.

The postconditions generated by the system rules will be the preconditions for the behavior rules and vice versa. Because we only consider the authority increasing effects of the interactions and of the influence the entities have on the interactions, the authority will never decrease in a KBM.

As part of the approximation strategy, KBMs neglect the authority-decreasing influence a change in behavior can have. We will provide alternative ways (Section 5.6.5) to investigate the effects of authority-decreasing behavior changes.

The fact that all rules in a KBM are monotonic (cannot decrease authority), has major consequences for the calculation of authority bounds in KBMs:

1. The calculation of authority in the KBM is a fix point calculation: it ends when no interactions can add additional authority (all executable system rules have been executed) and no entity's influence on the interactions can make more interactions possible (no behavior rules can make more system rules executable).
2. The fix point is computable in polynomial time corresponding to the size of the KBM.

To ensure that the KBM is a tractable and safe approximation of the software, even when the software potentially spawns infinitely many software entities, we will use a technique called *aggregation* (Section 5.3.1). That will allow us to model infinitely many software entities into a finite set of KBM subjects, while keeping the approximation safe.

Remark Note that KBMs will not only be used to prove safety properties in modeled software by calculating the fix point for reachable authority. A more complex and interesting application will use fix point calculations to find the necessary restrictions for the relied-upon subjects to guarantee the safety requirements and allow the development of a secure program by respecting these restrictions.

5.2.2 Refining Insufficiently Accurate Approximations

It is trivial to construct a tractable KBM that safely models all kinds of software: use a single subject with unrestricted behavior to model all entities. All authority will then be reached in the KBM. Clearly, safety and tractability are necessary, but not sufficient to solve the practical safety problems of a software engineer.

When a safety requirement is violated in a KBM (authority was reached that should not have been reachable), that constitutes no proof that the requirement is also violated in the software. It only proves that it is possible to implement software that is modeled by the KBM, in which the safety property will actually be violated.

To make our approach useful for software engineers, we will support progressive refinement of the modeled behavior (Section 5.6). It will be relatively easy to find out what behavior was responsible for the violation in the KBM. Then, if the software engineer concludes that the behavior of some KBM subject was not sufficiently refined to capture the actual restrictions of the programmed entities her security strategy relies upon, she can refine the modeled behavior of the subjects involved, without having to re-construct the whole KBM from scratch.

Section 5.5 gives an elaborate example of behavior refinement. Section 5.6 will then present the general technique for refinement and discuss several applications.

5.3 The Basic Elements of KBMs

In this section we describe the parts that constitute a KBM in greater detail. The code examples in figure 5.1 will be used to illustrate how every part of a KBM corresponds to an aspect of the software being modeled or designed. KBMs will be used in later sections to define safety problems.

Section 5.7 will give formal definition of KBMs as a mathematical structure and describe its logical semantics, which will be used to derive an upper bound for the authority that can be reached from an initial configuration. The declarative language SCOLL, that will be introduced in chapter 6 for the purpose of describing safety problems that are relevant to software engineers, will have a logical semantics that is expressed in terms of KBMs.

KBMs consist of four parts:

Subjects : A fixed and static set of subjects that each represent a software entity or an aggregate (set) of software entities

Predicates : A set of uniquely labeled relations, each with its own finite arity, that model the permissions and the authority in the software.

System Rules : A set of rules that model the different types of interactions that are possible in the software. These rules must respect the restrictions imposed by the protection system. Creation of new entities is a special form of interaction that, in some circumstances, can be neglected (see section 5.3.3).

Behavior Rules : A set of rules that model the dynamic influence the entities in the software have on the interactions.

Knowledge Behavior Models (KBMs) are named after the two types of authority that they can model:

- ***Knowledge*** is authority that represents a positive influence for which an entity is sensitive. It models what an entity can *know* (or sense) about its environment at runtime.

- ***Behavior*** is authority that represents a positive influence exerted by a entity upon the interactions.

Remark KBMs do not describe an initial configuration from which the safety analysis will start and neither do they describe requirements or expectations about the software. These elements will be added separately, when expressing the actual safety problems.

5.3.1 Subjects

We will use KBMs with a finite set of subjects to model the entities that are involved in the propagation of permissions and authority. For software engineers and programmers, these agents are the entities in their software. In general, the entities can best be identified by the following properties:

- Entities can have and use permissions and authority.
- Entities can interact with other entities.
- Entities can propagate permissions and authority by interacting with other entities.
- Entities can (often) create other entities.

For programs in a pure object oriented language, the object instances are the best candidates. For procedural languages, the entities are the runtime instances of the procedural closures. Multi paradigm languages contain both kinds of entities.

It is possible to analyze and model code at a higher level and consider modules or components to be the entities. The examples in this thesis usually focus on objects and procedures.

Because we want to calculate an upper bound on authority, it is crucial that all possible entities are modeled, including the ones that will possibly be generated in all possible executions of the software. Also, when modeling existing code, don't forget to consider the entities that may be hidden at first sight: loaded modules and classes, or classes and modules that are globally available at runtime, may contain other objects and procedures, or can themselves be entities as defined above.

In practice, the task of identifying the subjects that will model the software entities is not that complex or cumbersome. It is simplified considerably by the fact that a single subject can model a dynamic and potentially infinite set of entities in the software.

It is advisable to model all unknown entities together with all other unknown entities they can directly influence (access) and together with all entities they can create at runtime as a single subject.

The relied-upon entities should be easy to spot for the programmer, as they are the entities upon whose restricted behavior he relies to restrict the authority that will be reachable. They too can be combined, but caution will be necessary when modeling their behavior to make sure that the KBM remains a safe approximation of the software.

Notation

- Subjects are indicated in lowercase, e.g. `alice`
- Variables that range of the set of subjects are indicated in initial capital, e.g. $\mathtt{S}_1$.

Aggregating Multiple Entities into a Single Subject *Aggregation* is a technique to model a dynamic set of entities as a single subject, thereby making sure that:

- *Individual behavior implies aggregate behavior*: the conditions for the aggregated subject to interact in the model are weaker than the corresponding conditions to interact in the software for every entity in the set.
- *Individual authority implies aggregate authority*: the effects of the aggregated subject's interactions include the authority increasing effects of the corresponding interactions of all the entities in the set.
- *Aggregation of initial conditions*: the initial configuration contains the aggregated relations for every aggregated subject. For instance, if `alex` has initial access to `chris`, and `bill` has initial access to `dave`, aggregating `alex` with `bill` would give the aggregated subject initial access to both `chris` and `dave`.

These requirements will be explained further in section 5.3.4 and expressed formally in section 5.7.4. Corollary 2 of theorem 1 proves that the authority that is reachable in a KBM with aggregated subjects safely approximates (over-estimates) the authority that would have been reachable if all entities had been modeled as separate subjects.

For now it suffices to know that aggregation of any set of entities into a subject with unrestricted behavior is simple and safe: adding more entities to an aggregation cannot make the unrestricted behavior of the subject even more unrestricted. We only have to remember to model the initial relations (e.g. access permissions) between any of these entities and the entities that are modeled in other subjects. To model the initial access among the aggregated entities it suffices to give the aggregated subject access to itself.

Example Figure 5.1 is already a simple example of aggregation. The set of subjects that will model all entities in the software is: {`alice`, `bob`, `carol`}. These three subjects must represent between them the complete (unbounded) set of entities in the software, including those that can possibly be created at runtime.

The most natural (and default) aggregation strategy is to map all created entities with their creator. Subject `bob` then models the entity that was immediately created from the module "bobModule" and all entities created by it and by their descendants.

In the example we also aggregate `bob` with his creator, the "BobModule", and with all other entities that may be hidden or created by that module. Subject `carol` is aggregated in a similar way. Notice in the access graph of figure 5.1 that `bob` and `carol` each have access to themselves. It is clear from the code in the example that subject `alice` does not create any entities: `alice` will represent herself only.

The default aggregation strategy is most useful to avoid having to model creation at all: all subjects already aggregate all offspring of the entities they represent.

Aggregation Strategies Depending on the situation, many interesting aggregation strategies can be considered:

1. The default aggregation strategy: model all child entities with their parent.
2. Model all entities of a similar type (with the same behavior) as a single subject.
3. Group all entities that have a certain initial permission (e.g. access to a specific resource).

4. Group all entities that were not modeled by other subjects into a `remaining` subject.

5. Arbitrary grouping, as long as all possible entities are modeled.

A detailed example of a particularly useful and interesting application of aggregation by subject behavior will be described in section 6.9 and analyzed in depth in section 8.3.1.

5.3.2 Predicates and Facts

In KBMs, all permissions, authority, and behavior will be expressed using n-ary predicates over the subjects. This is a direct extension of the binary protection matrix in the HRU models (Section 3.1) and of the (also binary) graph-based representation of permissions and authority in Take-Grant models (Section 3.2).

Technically, a predicate of arity n is the combination of a label `p`, with a complete function that maps n-tuples (of subjects) to $\{true, false\}$. For an n-ary predicate with label `p` the labeled n-tuples of subjects, $\texttt{p(s}_1,\ldots,\texttt{s}_n\texttt{)}$ will be called that predicate's facts.

We say that a fact $\texttt{p(s}_1,\ldots,\texttt{s}_n\texttt{)}$ is *true* if the n-ary predicate named `p` is true for the tuple $(\texttt{s}_1,\ldots,\texttt{s}_n)$ (if the tuple is in the relation defined by the predicate).

Notation

- All facts $\texttt{p(s}_1,\ldots,\texttt{s}_n\texttt{)}$ specify a relation of the first subject in the fact, $\texttt{s}_1$, towards the other subjects $\texttt{s}_2 \ldots \texttt{s}_n$. The first argument is called the base subject.

- Predicates are presented in the same way as their facts: $\texttt{p(S}_1,\ldots,\texttt{S}_n\texttt{)}$, but with capitals letters to indicate that they range over variables, not over constants.

- When appropriate, we will use an alternative notation for predicates and facts that express subject behavior or subject knowledge (not permissions). We position the first argument in front of the label, followed by a colon. Instead of the normal notation $\texttt{b(S}_1,\ldots,\texttt{S}_n\texttt{)}$, we write: $\texttt{S}_1\texttt{:b(S}_2,\ldots,\texttt{S}_n\texttt{)}$.

  This notation underlines the fact that it is $\texttt{S}_1$'s behavior or knowledge we are talking about. The alternative notation is not used for permissions, because permissions are managed only by the protection system.

Three kinds of predicates will be used in a KBM:

1. **Permission predicates :** model the ways in which an entity is allowed to interact with other entities.

2. **Behavior predicates :** model the ways in which an entity can influence its interactions with other entities.

3. **Knowledge predicates :** model the ways in which an entity can be influenced.

We discuss each of these predicates using examples that correspond to figure 5.1.

1. Permission predicates model if one entity is allowed to interact with other entities in a certain way. In most protection systems permissions are binary and their label corresponds to a right the first entity holds over the second entity. In memory safe programming languages like Oz and Emily, the only permission is the permission to *access* an entity and is implied by the reference to that entity that acts as a capacity, because it is unforgeable. We will use the binary predicate labeled *access* to model this permission.

In the example in figure 5.1, the *initial* permissions are:

- `access(alice,bob)`
- `access(bob,bob)`
- `access(carol,carol)`

2. Behavior predicates model how an entity will positively influence its interactions with other entities. In the invoke-return model of our example (Figure 5.1), entities have a few basic ways to influence the interactions that are allowed by their permissions:

1. Choose the entities they will invoke.
2. When invoking an entity: choose the input argument(s) for the invocation.
3. When invoking an entity: choose if they will accept return values from the invocation.
4. When being invoked, choose if they want to accept input arguments.
5. When being invoked, choose if and what values they want too return

These choices can be modeled with the following four predicates:

- `may.sendTo(A,B,C)` : subject `A` chooses to invoke subject `B`, and thereby emit (propagate) subject `C` as an input argument of the invocation.
  Alternative notation: `A:may.sendTo(B,C)`.
- `may.getFrom(A,B)` : subject A chooses to invoke subject `B` and then accept `B`'s return values form the invocation.
  Alternative notation: `A:may.getFrom(B)`.
- `may.receive(B)` : subject `B`, when being the responder to an invocation, chooses to accept input arguments.
  Alternative notation: `B:may.receive()`.
- `may.return(B,D)` : subject `B`, when being the responder to an invocation, chooses to emit (return) subject `D` to the invoker.
  Alternative notation: `B:may.return(D)`.

We used a prefix `may.` in all these behavior predicate labels. The motivation for this prefix is to remind the reader of the fact that we are modeling possible behavior. This convention is not imperative. All predicate labels can be chosen freely by the KBM modeler.

Notice that we do not model the entity's choice to invoke an entity without providing input arguments or accepting return values. Therefore, in our example, the authority that can be conveyed by invoking an entity in that way must always be implied by the access permissions. This is only relevant when we want to analyze the propagation of data (Section 5.6.3).

Notice also that we did not model the entity's choice to create new entities. In the example of figure 5.1, we must therefore assume that all subjects, including `alice`, do indeed consistently choose to create new entities. That is a crude over-approximation of `alice`'s actual behavior, that could in principle make our approximation too crude to be useful. We will skip this issue for now and return to it in section 5.4.

Remark *Contrary to choices 1 to 3, choices 4 and 5 don't require the entity to have a permission. The invoke-return model is a special case of the simple invoke model, using **continuations**. Consider that the invoker always provides an extra argument, referencing a continuation of himself, that behaves exactly as the invoker when he will have accepted the return value. Access to that continuation is a permission, provided by the invoker, for the invokee to use when returning a value.*

3. Knowledge predicates model the influence that is exerted upon an entity. That influence can originate from a successful interaction in which the entity was involved, or from the entity itself, without any interaction.

In KBMs, knowledge, like behavior, is never shared between entities. Successful interactions may provide different knowledge to different participants in the interaction. For instance, when `alice` invokes `bob` and gets `carol` as a return value, `alice` knows who provided `carol` to her, but `bob` does not know to whom he returned `carol`.

We distinguish two types of knowledge predicates:

(a) Knowledge about successful interaction. This type of knowledge models the direct influence a successful interaction has on an entity, usually an entity that played a certain role in the interaction. In our example of figure 5.1, successful invoke-return interactions can be modeled in four predicates:

- `did.sendTo(A,B,C)` : subject `A` chose to invoke subject `B` with argument `C` and knows that it succeeded.
  Alternative notation: `A:did.sendTo(B,C).`
- `did.getFrom(A,B,D)` : subject `A` chose to invoke subject `B` and to accept `B`'s return values and knows that it succeeded. The returned value is `D`.
  Alternative notation: `A:did.getFrom(B,D).`
- `did.receive(B,C)` : subject `B` chose to accept input arguments when invoked and knows that such an interaction succeeded. The accepted input argument is `C`.
  Alternative notation: `B:did.receive(C).`
- `did.return(B,D)` : subject `B` chose to return subject `D` and knows that such an interaction succeeded.
  Alternative notation: `B:did.return(D).`

In most KBMs the knowledge predicates will correspond directly to the behavior predicates.

We used a prefix `did.` in all these knowledge predicate labels. The motivation for this prefix is to remind the reader of the fact that we are modeling a guaranteed effect of successful behavior. This convention is not imperative. All predicate labels can be chosen freely by the KBM modeler.

(b) Private knowledge predicates. This type of knowledge is used to approximate the internal state of a specific subject. Private knowledge is generated by the subject itself and is not visible to any other subject (all inter-subject influence is conveyed by interaction).

Private knowledge predicates can be used to model all kinds of relations of which an entity keeps track. For instance, the fact that a relied-upon object stores certain entities in a particular instance variable can be seen as a relation the object has towards that entity.

5.3.3 System Rules

The different types of interactions that are possible in the software, are modeled in a KBM as a set of system rules. Every system rule is a set of preconditions and a set of postconditions.

The preconditions model the contribution of the subjects involved in the interaction and the requirements demanded by the protection system, for the interaction to be possible. The conditions concerning the contribution of the subjects are expressed as behavior predicates. The conditions imposed by the protection system are modeled as permission predicates.

The postconditions model the authority that is propagated upon successful interaction. The postconditions are permission predicates and knowledge predicates.

Notation System rules are expressed as implications of the following form:

$$\mathtt{p}_1(\mathtt{S}_{1,1},\ldots,\mathtt{S}_{1,ar(p_1)}) \wedge \ldots \wedge \mathtt{p}_k(\mathtt{S}_{k,1},\ldots,\mathtt{S}_{k,ar(p_k)})$$
$$\Rightarrow \mathtt{p}_{k+1}(\mathtt{S}_{k+1,1},\ldots,\mathtt{S}_{k+1,ar(p_k+1)}) \wedge \ldots \wedge \mathtt{p}_m(\mathtt{S}_{m,1},\ldots,\mathtt{S}_{m,ar(p_m)})$$

Example For our running example (Figure 5.1), we model two rules for the invoke-return interactions: one for each direction in which the authority can propagate. Notice that we use the alternative notation for behavior and knowledge.

```
sys1. access(A,B) ∧ access(A,X) ∧ A:may.sendTo(B,X)
      ∧ B:may.receive()
      ⇒ access(B,X) ∧ A:did.sendTo(B,X) ∧ B:did.receive(X)

sys2. access(A,B) ∧ access(B,Y) ∧ A:may.getFrom(B)
      ∧ B:may.return(Y)
      ⇒ access(A,Y) ∧ A:did.getFrom(B,Y) ∧ B:did.return(Y)
```

Rule sys1 models interaction whereby `A` invokes `B` with the argument `X` and `B` accepts input values. The $access$ permissions indicate that `A` should have access to `B` and to `X`, before the interaction is allowed by the protection system. Successful interaction will result in one new permission (`B` gets access to `X`) and new knowledge communicated to `A` and `B` about the result of the interaction.

Rule sys2 models interaction whereby `A` invokes `B` and `B` returns `Y`. The $access$ permissions indicate that `A` should have access to `B` and `B` should have access to `Y`, before the interaction is allowed by the protection system. Successful interaction will result in one new permission (A gets access to Y) and new knowledge communicated to A and B about the result of the interaction.

Notice that two pairs of complementary roles can be identified for the subjects involved in an interaction.

1. Invoker and Responder: the former invokes the latter.
2. Emitter and Collector: the former propagates access to the latter.

Each of the roles in every pair must be present in every interaction, but the combination of roles from every pair can differ: the invoker can be emitter if the responder is collector and vice versa.

Remark Notice that these rules match the requirements for a capability system, of sections 4.3.4 and 4.3.5:

- The emitter *chooses* which subject will be propagated.
- The invoker *chooses* which subject it will invoke.
- The invoker *chooses* in which circumstances and to what effect it will use its capabilities.
- The responder *chooses* in which circumstances it will cooperate with the invoker in realizing what authority.

5.3.4 Behavior Rules

Earlier, we modeled the "willingness" of an entity to cooperate in an interaction, using predicates. That alone would allow us already to offer categories of subjects with static behavior (like the active and passive subjects in the Take-Grant systems of section 3.2).

But, since we were looking for more expressive power, we will use behavior rules to express the conditional willingness of an entity to cooperate in an interaction. Behavior rules will allow us to express what it is about a programmed entity, we rely on.

Like system rules, behavior rules consist of a set of preconditions and a set of postconditions. The preconditions are knowledge predicates that model the conditions for the entity to be "willing to interact".

The postconditions will consist of behavior predicates that model the way in which the entity may interact if the preconditions are satisfied. In more expressive instances of KBMs, the postconditions of a behavior rule can also contain private knowledge predicates that model changes in the internal state of an entity. We will see an example of that in section 5.4.

Behavior rules express the conditional behavior of a single subject, in terms of predicates that refer to that subject's authority (knowledge and behavior). Therefore, the first element in every predicate of a behavior rule must refer to the subject whose behavior the rule defines.

Notation Behavior rules are expressed as implications of the following form:
$\mathtt{p}_1(\mathtt{s}, \mathtt{s}_{1,2}, \ldots, \mathtt{s}_{1,ar(p_1)}) \wedge \ldots \wedge \mathtt{p}_k(\mathtt{s}, \mathtt{s}_{k,2}, \ldots, \mathtt{s}_{k,ar(p_k)})$
$\Rightarrow \mathtt{p}_{k+1}(\mathtt{s}, \mathtt{s}_{k+1,2}, \ldots, \mathtt{s}_{k+1,ar(p_k+1)}) \wedge \ldots \wedge \mathtt{p}_m(\mathtt{s}, \mathtt{s}_{m,2}, \ldots, \mathtt{s}_{m,ar(p_m)})$
Notice that the first element in the predicates is now a subject (s), instead of a variable:

it is the subject who's behavior is expressed in that rule. In behavior rules we will usually use the alternative notation for behavior and knowledge predicates.

Unrestricted Behavior In our running example (Figure 5.1) we identified two subjects, `bob` and `carol`, about whose behavior we did not know anything for sure. To be safe, we must assume that these subjects contribute to maximize the authority that can be reached in the software. Therefore we model their behavior with a single rule that has an empty set of preconditions and has all the subject's behavior predicates in the postconditions.

For instance, `bob`'s behavior will be expressed with the following single behavior rule:

```
⇒ bob:may.sendTo(B,X) ∧ bob:may.getFrom(B)
  ∧ bob:may.receive() ∧ bob:may.return(Y)
```

This rule expresses that `bob` is willing to interact unconditionally, with all entities in all possible ways.

If we would give every subject unrestricted behavior, only the permission predicates in the system rules would remain actual restrictions. We could already calculate a permission based bound on authority. For some applications, permissions based calculation of authority bounds can suffice, but usually the calculated bound will be too big, because the restricting influence of the relied-upon entities was not modeled.

Restricted Behavior The behavior of `alice` in our example (Figure 5.1), is modeled by the following two rules:

1. `⇒ alice:may.receive()`

2. `alice:did.receive(X) ⇒ alice:may.sendTo:(bob,X)`

The first rule indicates that `alice` is willing to accept input arguments unconditionally, when given the choice. The second rule states that `alice` will invoke `bob` with the argument `X`, on condition that she has received that argument `X` in an interaction in which she was invoked (she was the responder).

Notice that, in the predicate on the right hand side of the second rule, not only the first element is a subject, but so is the second: `bob`. That is OK, because, in figure 5.1 it is clear that `alice` only invokes `bob`. However, in general, a variable that is referenced in the source code of a relied-upon entity, may, at different steps in different executions, contain entities that are modeled as different subjects. Therefore, the practice of using subjects in behavior rule predicates, other than for the first argument of the predicate, is strongly discouraged. In SCOLL, it will be forbidden.

The alternative approach is to model `alice`'s relation to the subjects in the variable `Bob` as a private knowledge predicate, for instance the binary predicate `isBob()`. The second rule of `alice`'s behavior will then be expressed as:

```
2. alice:did.receive(X) ∧ alice:isBob(B)
   ⇒ alice:may.sendTo(B,X)
```

Later, when we model the initial configuration from which the calculation should start, the private knowledge of `alice` will be initialized to: `alice:isBob(bob)`.

5.4 A Simple Model for Object-Capabilities with Creation

Using a straight forward mapping of the mechanism to propagate authority in object capabilities (Section 4.3), we will now derive a set of simple knowledge and behavior predicates, and a set of system rules to construct a more realistic model for capabilities that includes an explicit model for entity creation.

Table 5.1 lists the predicates we encountered in section 5.3, be presented with their arity. We will keep the system rules for invocation-based interaction that were introduced section 5.3 and only discuss the additional system rules that govern creation.

Table 5.1: Overview of the predicates introduced in section 5.3

.

	predicate label with arity
behavior predicates	`may.sendTo/3`
	`may.getFrom/2`
	`may.receive/1`
	`may.return/2`
knowledge predicates	`did.sendTo/3`
	`did.getFrom/3`
	`did.receive/2`
	`did.return/2`

5.4.1 Running Example

To illustrate how we model the creation of a relied-upon entity by another relied-upon entity, we introduce new code examples in figures 5.2 and 5.3

Figure 5.2 shows the Oz code for an object `AliceObj`, and two unspecified entities `Bob` and `Carol`. Figure 5.3 shows an equivalent situation in Emily.

The Subjects and the Initial Access Graph

Again, we will assume that the runtime environment for the software does not provide ambient authority. We will aggregate the unknown entities like before, together with the module they were created from, and with all their possible offspring. We model the remaining code as two subjects: `alice`, corresponding to `AliceObj` in Oz (`aliceObj` in Emily), and `proxy`, corresponding to all the entities that are created by `alice`.

The fact that `proxy` is an object in Oz, and a function in Emily, is irrelevant. In oz, Objects can implement proxies by using the `otherwise` method, that handles methods that are not explicitly implemented. Emily (Ocaml) has no similar construct. For the purpose of analyzing the propagation of authority, these details about the interaction mechanism are unimportant, because we consider an abstract interaction model.

We will consider the initial situation, at a stage where `alice` did not yet create a `proxy`. The initial graph will nevertheless contain the `proxy` subject, to represent the complete set of subjects. Because `proxy` is not yet created, it is not connected to the other subjects in the access graph.

```
declare
   Bob = {{Link ["unknownModule.ozf"]}.1.makeInstance}
   Carol = {{Link ["carolModule.ozf"]}.1.makeCarol}

   AliceObj = {New class $
                   attr precious
                   meth init(protect: P)
                      {Wait P}     % P must be bound
                      @precious =  P
                   end
                   meth getPrecious($)
                      {Wait @precious}     % must be bound
                      @precious
                   end
                   meth makeProxyFor(Client)
                      {Wait @precious}     % wait for
initialization
                      {Wait Client}     % must be bound
                      Target = @precious
                      Proxy = {New class $
                                     meth init() skip end
                                     meth otherwise(Method)
                                        {Target Method}
                                     end
                                  end
                               init()}
                   in
                      {Client useProxy(Proxy)}
                   end
                end
             init(protect: Carol)}

   {AliceObj makeProxyFor(Bob)}
```

Figure 5.2: `bob`, `carol` and the relied-upon subject `alice` in Oz

```
   let bob = BobModule.makeBob();
   let carol = CarolModule.makeCarol();

   let aliceObj  = object
                      val precious = carol
                      method getPrecious = precious
                      method makeProxyFor client =
                         let proxy input = precious input;
                         ignore (client#useProxy proxy)
                   end;

   alice#makeProxyFor bob;
```

Figure 5.3: `bob`, `carol` and the relied-upon subject `alice` in Emily

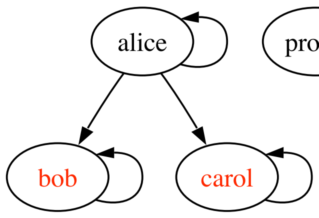


Figure 5.4: The initial access graph

The other three subjects all have access to themselves, including `alice`, even if we know from `alice`'s code that she does not refer to herself. In many programming environments, objects can have access to themselves if they want to, by using the pseudo variable **`self`**.

The access graph of figure 5.4 depicts the initial access permissions of the subjects. Notice that KBMs will allow us to describe the initial state of the subjects in greater detail than is suggested in this figure. It can include all kinds of initial conditions to model the initial state of the entities more accurately.

For instance, in our running example, the initial conditions will also include the facts `alice:init(carol)` and `alice:makeProxyFor(bob)` to indicate that `alice`, in her initial state, not only has access to `carol` and `bob`, but also has a specific relation to each of them, that will influence if and how `alice` will interact with either of them. Since `proxy` does not yet "exist" in the initial configuration we choose to start from, no initial knowledge is provided to it yet. The necessary access and knowledge will be provided to `proxy` by its creator (`alice`).

Behavior

For the same reasons as in the previous example, the behavior of the subjects `bob` and `carol` will be unrestricted.

The code of `AliceObj` is now larger and more complex. Trying to detect *alice*'s behavior restrictions directly is no longer advisable, because of the possibility of errors. It is simpler to model what `alice` does than what she doesn't do: the former can be derived from the individual methods of `AliceObj`.

1. the complete effect of invoking the `init()` method can be modeled with two private knowledge predicates, in a single behavior rule:
   `alice:init(P)` ⇒ `alice:precious(P)`

2. the effect of the `getPrecious()` method, is expressed in the behavior rule:
   `alice:precious(P)` ⇒ `alice:may.return(P)`

3. to model the effect of `makeProxyFor()` we have to model subject creation. The next sections will explain in detail how we do that, but the intuition can be given in two simple rules:

   (a) `alice` gets access to `proxy`, by the act of creation.

   (b) `alice` can convey authority to `proxy`, without having to invoke one of `proxy`'s methods, by providing variables in `proxy`'s outer scope. In this case, `proxy` is endowed with access to the variable `ProxyTarget`, which is bound to the value in the attribute `precious`.

These are exactly the rules for parenthood and endowment in capability systems, we saw in section 4.3.3. Anticipating the definition of the predicates we will use for parenthood and endowment, we can express `alice`'s endowment behavior as:
`alice:precious(P)` $\wedge$ `alice:did.create(Child)`
$\Rightarrow$ `alice:may.endow(Child,P)`

The `makeProxyFor()` method will then convey the newly created entity to the `Client` argument, after having made sure that the argument is bound. This can be expressed with another private knowledge predicate `makeProxyFor/2`, and the behavior rule:
`alice:makeProxyFor(Client)` $\wedge$ `alice:did.create(Child)`
$\Rightarrow$ `alice:may.sendTo(Client,Child)`

The behavior for the `proxy` subject, representing all entities that are created by `alice`, can be derived in a similar way from the individual methods of its anonymous class. To model `proxy`'s behavior it is not necessary to look outside the class definition, not even to include the outer scope of the class definition.

1. The `init()` method has no effect. `proxy` will get its initial access and knowledge via endowment by its creator `alice`, not by invoke-return style interaction.

2. The maximal effect of invoking any other method (represented by the argument `Method` in the method `otherwise()`), can be modeled using one private knowledge predicate: `target/2`, and four behavior rules:

   (a) $\Rightarrow$ `proxy:may.receive()` : `proxy` accepts input variables.

   (b) `proxy:target(T)` $\wedge$ `proxy:did.receive(X)`
   $\Rightarrow$ `proxy.may.sendTo(T,X)` : `proxy` forwards input variables to its target.

   (c) `proxy:target(T)` $\Rightarrow$ `proxy:may.getFrom(T)`:
   `proxy` invokes its target and accepts the return values of that invocation.

   (d) `proxy:target(T)` $\wedge$ `proxy:did.getFrom(T,X)`
   $\Rightarrow$ `proxy:may.return(X)` :
   `proxy` returns the values that it collected by invoking its target.

   Rules (c) and (d) need some explanation. When modeling `proxy`'s behavior from the Oz code in figure 5.2, it may not be obvious that `proxy` also propagates values from its target to its invokers. That is because of the logical variables in Oz. The arguments of the method `Method` that is forwarded by `proxy` to its target, can be unbound logical variables. When the target binds a logical variable, the propagation of the value goes in the opposite direction : from `proxy`'s target to its invoker.

5.4.2 Predicates for Subject Creation

Parenthood and Endowment To model creation of new entities we distinguish the *parent* and *child* roles. We will allow a parent to pass on its access to its children without the need for the child to give its consent. This model fits the mechanism in a lexically scoped language, in which newly created (inner) closures have directly access to the creating (outer) scope. In accordance with section 4.3.3, we call this *endowment* of the child by its parent.

In many systems, including capability systems, the parent automatically gets a reference to its created child. This will be referred to as *parenthood* (also in Section 4.3.3).

Aggregation and the Parent-Child Relation Aggregation allows us to map all entities, the ones in the initial configuration and the ones that can possibly be created, into a finite set of subjects in the model. When mapping the parent-child relation between the entities onto the subjects, the hierarchical nature of the parent-child relation can be lost. That can lead to situations where one subject is another one's child and parent at the same time. More often though, a subject will model a complete tree of descendants. We will prove in section 5.7 that aggregation respects the safety properties: if a more-aggregated model is safe, all less-aggregated models will be safe too.

The Pseudo Permission Predicate: `child/2` Just like the set of subjects, the aggregated parent-child relation between the subjects is a fixed and static part of the KBM. That relation is expressed using the predicate `child/2` and is guarded by additional system rules. Since the static parent-child relation between the subject is an artifact of aggregation, we will call the `child/2` predicate a “pseudo permission”.

Table 5.2: The pseudo permission predicate `child/2`

predicate	comments
child(S_1,S_2)	subject S_1 models entities that possibly create other entities that are modeled by subject S_2

Behavior Predicates Table 5.3 lists the behavior predicates for creation and endowment.

The behavior predicate `may.create/2` expresses the parent's intention to create a child. The behavior predicate `may.endowWith/3` expresses the parent's intention to endow its created child with another entity.

Table 5.3: Behavior Predicates for Parenthood and Endowment

predicate	comments
S_1:may.create(S_2)	S_1 intends to create S_2
S_1:may.endowWith(S_2,X)	Parent S_1 endows its child S_2 with access to X

Knowledge Predicates Table 5.4 lists the knowledge predicates for creation and endowment.

- `did.create/2` informs the parent subject that creation of a child has happened.
- `did.endowWith/3` informs the parent subject that it has endowed a child with access to an entity.
- `was.endowedWith/2` informs the child subject that creation it was endowed by a parent with access to an entity. This can be useful to express that a child entity is more collaborative in its behavior towards entities it was endowed with.

Table 5.4: Knowledge Predicates

predicate	comments
S_1`:did.create(`S_2`)`	parent S_1 created S_2
S_1`:did.endowWith(`S_2`,X)`	parent S_1 endowed its child S_2 with `X`
S_2`:was.endowedWith(X)`	child S_1 was endowed with `X`

Notice that we do not provide knowledge about the parent to the child. It is left to the parent's discretion, whether or not it will endow the child with access to the parent, provide this access via invocation, or not at all.

Important Remark : The reader should always keep in mind that we model only a safe approximation of the actual behavior of the entities, modeled as subjects. For instance, S_1`:may.create(`S_2`)` actually means: what we know about (and model from) the entities modeled by subject S_1, does not exclude the possibility that at least one of these entities creates a new subject that is modeled as subject S_2.

5.4.3 System Rules for Subject creation

System rules specify when behavior is relevant and effective, and what its effects can be. We have encountered the system rules that govern the propagation of authority via invoke-return style interaction in section 5.3.3.

To prevent the subjects from modeling impossible creation and endowment (as opposed to illegal interactions prevented by the protection system) we add the following system rule that will check the fixed parent-child relation between the subjects in the KBM.

sys3. `P:may.create(C)` $\wedge$ `child(P,C)`
$\Rightarrow$ `access(P,C)` $\wedge$ `p:did.Create(C)`

sys4. `P:did.create(C)`$\wedge$ `access(P,X)` $\wedge$ `P:may.endowWith(C,X)`
$\Rightarrow$ `access(C,X)` $\wedge$ `P:did.endowWith(C,X)`
$\wedge$ `C:was.endowedWith(X)`

Rule sys3 models *parenthood.* If a subject `P` wants to create a subject `C` and there is a pre-established parent-child relation in the KBM between `P` and `C`, this rule will give the parent *access* to its child upon creation and notify the parent of the creation. The `child/2` predicate on the left hand side enforces the static parent-child relation between the subjects. Section 5.4.6 will show how the `child/2` relation can be specified as part of the initial configuration.

Rule sys4 models *endowment*. If a subject `P` has created a subject `C`, and `P` has access to `X` and wants to endow that access to `C`, `C` will get access to `X` and both `C` and `P` will be informed about the endowment. It is clear that `C`'s behavior has no influence, because no behavior predicate in the left hand site of the rule contains `C` as its first argument.

5.4.4 Behavior Rules for Creation

The predicates for endowment and creation can be used in the following ways by the subjects:

- By the parent, to specify the conditions in which he intents to create a child entity, and/or endows a created entity with access to another entity.
- By the parent, to express behavior activated by the creation or the endowment of an entity.
- By the child, to express behavior that depends on the entities it was endowed with.

In our running example, `alice`'s creation of proxies can be modeled by the following simple behavior rules:

1. $\Rightarrow$ `alice:may.create(C)` : `alice` creates new entities
2. `alice:precious(P)` $\Rightarrow$ `alice:may.endowWith(C,P)` :
   `alice` endows her children with access to the entities that are "precious" to her.

5.4.5 The KBM of the running example

Table 5.5 gives an overview of the KBM of our running example.

5.4.6 The Initial Configuration

The analysis in a KBM starts from a configuration that describes the permissions and initial state of the entities. These conditions can have been generated by out-of-context mechanisms (see Section 4.3.2), but can also reflect the conditions after a partial evolution from a previous configuration.

All initial conditions will be expressed as permission facts and knowledge facts. Permission facts (and pseudo-permissions) will be used to represent the initial configuration of permissions and the fixed parent-child relation between the subjects of the KBM. Private knowledge is most useful to represent the initial internal state of a subject in the configuration.

Non-private knowledge facts can be used to reflect the knowledge a subject has acquired from interactions before the initial state.

In short, the initial configuration has to provide information about:

- The initial *access* permissions between the subject
- The aggregated parent-child relation

Table 5.5: The KBM of the running example

subjects	
relied upon	unknown
`alice` `proxy`	`bob` `carol`

predicates			
permission	behavior	knowledge	private knowledge

`access/2`
`child/2`
	`may.sendTo/3`
	`may.getFrom/2`
	`may.receive/1`
	`may.return/2`
	`may.create/2`
	`may.endowWith/3`
		`did.sendTo/3`
		`did.getFrom/3`
		`did.receive/2`
		`did.return/2`
		`did.create/2`
		`did.endowWith/3`
		`was.endowedWith/2`
			`alice:`
			` init/2`
			` target/2`
			`proxy:`
			` target/2`

system rules
access(A,B) ∧ access(A,X) ∧ A:may.sendTo(B,X) ∧ B:may.receive() ⇒ access(B,X) ∧ A:did.sendTo(B,X) ∧ B:did.receive(X)
access(A,B) ∧ access(B,Y) ∧ A:may.getFrom(A,B) ∧ B:may.return(Y) ⇒ access(A,Y) ∧ A:did.getFrom(B,Y) ∧ B:did.return(Y)
P:may.create(C) ∧ child(P, C) ⇒ access(P,C) ∧ P:did.create(C)
P:did.create(C)∧ access(P,X) ∧ P:may.endowWith(C,X) ⇒ access(C,X) ∧ P:did.endowWith(C,X) ∧ C:was.endowedWith(X)

behavior rules
alice:init(P) ⇒ alice:precious(P)
alice:precious(P) ⇒ alice:may.return(P)
alice:makeProxyFor(Client) ∧ alice:did.create(Child) ⇒ alice:may.sendTo(Client,Child)
⇒ alice:may.create(C)
alice:precious(P) ⇒ alice:endowWith(C,P)
⇒ proxy:may.receive()
proxy:target(T) ∧ proxy:did.receive(X) ⇒ proxy:may.sendTo(T,X)
proxy:target(T) ⇒ proxy:may.getFrom(T)
proxy:target(T) ∧ proxy:did.getFrom(T,X) ⇒ proxy:may.return(X)
⇒ bob:may.sendTo(B,X) ∧ bob:may.getFrom(B) ∧ bob:may.receive() ∧ bob:may.return(Y) ∧ bob:may.endowWith(C,X)
⇒ carol:may.sendTo(B,X) ∧ carol:may.getFrom(B) ∧ carol:may.receive() ∧ carol:may.return(Y) ∧ carol:may.endowWith(C,X)

- The initial state of each of the subjects: their private knowledge
- The "left-over" knowledge, of a previous model.

The initial configuration of the running example is listed in table 5.6.

Table 5.6: The initial configuration of the running example (Section 5.4.1).

permission facts	`access(alice,alice)`
	`access(alice,bob)`
	`access(alice,carol)`
	`access(bob,bob)`
	`access(carol,carol)`

parent - child relation	`child(alice,proxy)`
	`child(bob,bob)`
	`child(carol,carol)`

internal state	`alice:init(carol)`
	`alice:makeProxyFor(bob)`

previous evolution	

5.4.7 Modeling a Proof-of-Access Tester

The KBM model we have described here is dynamic and allows developers to express conditional behavior in the subjects, based on what can become observable to an entity about its relations with other entities. Still, the conditional behavior may not be as refined as we want it to be for practical safety analysis.

Suppose for instance that we that we can rely on the following knowledge about `carol`, in our running example: when invoked with an input argument, `carol` always checks if the received entity is identical to a predefined, passive entity that functions as a secret token. Only when the input argument matches the secret token, `carol` is willing to return her own secret. In the other case, she returns a less important, public value.

Doing so, `carol` effectively authenticates her invokers, not by identifying them (there is no need for that and she may not have access to them), but by comparing the input argument to a shared secret. To persuade `carol` to return her own secret, `carol`'s invokers have to prove to her that they have previous access to the shared secret. The code for the relied-upon `carol` we described here, is presented in figure 5.5.

The identity comparison test can be modeled in `carol`'s behavior as an initial private knowledge predicate $isSecretTkn/2$. In the current KBM, our best, safe, approximation is to express `carol`'s behavior as follows:

```
⇒ carol:may.receive()                          (1)
carol:isPublic(Y) ⇒ carol:may.return(Y)        (2)
carol:did.receive(X) ∧ carol:isSecretTkn(X)
⇒ carol:may.return(Y)                          (3)
```

In the initial configuration, we will add the necessary facts to let `carol` know what subject represents the secret token (`carol:isSecretTkn(secret)`) and what value(s) are public (`carol:isPublic(public)`).

```
PublicValue = {NewName}
SecretToken = {NewName}

Carol =  local
            SecretValue = {NewName}
            IsSecretToken = fun{$ Token}
                               Token == SecretToken
                            end
         in {New class $
                    meth init() skip end
                    meth get(Token $)
                       if {IsSecretToken Token}
                       then [SecretValue PublicValue]
                       else [PublicValue]
                       end
                    end
                 end
              init()}
           end
```

– `carol` as a relied-upon proof-of-access tester in Oz-E. –

```
let publicValue = object end;
let secretToken = object end;

let carol = object
               val secretValue = object end;
               val isSecretToken
                   = function token
                     -> token = secretToken
               method get token =
                  if isSecretToken token
                  then [secretValue; publicValue]
                  else [publicValue]
            end;
```

– `carol` as a relied-upon proof-of-access tester in Emily. –

Figure 5.5: `carol` as a relied-upon proof-of-access tester in Oz and Emily.

We did a very poor job, modeling alice's behavior. The actual access tester, described in the code of figure 5.5, can be relied upon to be much more restricted than we can express in our model. The real entity can accept a token, check it, return the secret value of the invoker that provided the secret token, and afterwards repeat the whole thing over again.

But in our current KBM, carol will, after the first time she is invoked with the correct secret token, return all her capabilities to all invokers, because she has no information about the invocation context. Our KBM can safely approximate carol's behavior, but not accurately enough to be useful for actual safety analysis. To solve this problem, the following section will introduce a refinement of the behavior predicates in our KBM.

5.5 Refining `may.return/2`

In this section we introduce two new behavior predicates to refine the may.return/2 predicate.

The refinement takes into account that, in the the invoke-return model, authority can propagate in both directions in the same invocation and the responder can check the input before deciding his return behavior for that invocation. The new behavior predicates will complement the original predicates rather than replacing them.

When refining behavior predicates, we will always make sure that the semantics of the original predicates remain unchanged. That will allow us to reuse the original system rules and to keep the original behavior rules for those subjects whose behavior we do not want to refine. The effort that was put into modeling a safe KBM will not have been in vain, should it turn out that the behavior approximation was too crude, because refining behavior in a KBM is an incremental process.

This section will conclude with two examples of the extra expressive power that can by gained by this particular form of refinement. In the next section we will generalize the refinement of behavior predicates.

5.5.1 Refined Predicates

The first refined predicate for may.return/2 is may.returnFor/3. It expresses the behavior of a responder who conditionally return a subject in the same invocation it received a certain subject in. It allows carol to express conditional responder behavior carol:may.returnFor(X,Y) that indicates her willingness to return a subject Y in the same invocation in which the invoker sent her at least one subject X.

The second refinement, may.returnFor0/2, is used to express the complementary behavior. B:may.returnFor0(Y) indicates the responder B's intention to return Y in those invocations in which the invoker emitted nothing. The importance of this complementary refined behavior predicate will become clear in section 5.5.5.

Table 5.7 explains the new predicates and their corresponding subject knowledge predicates.

We express the refinement relations formally in table 5.8. A complete formal account will be given in section 5.7 for the generalized strategy of refining behavior.

Note that, in table 5.8, the implications for the refined knowledge predicates point in the direction opposite to the implications for the refined behavior predicates.

Refined behavior predicates indicate behavior that is more specific than the original unrefined behavior. In this case, the refined behavior depends on detectable conditions

Table 5.7: Predicates for refining responder behavior.

predicate	comments
refined behavior predicates	
`B:may.returnFor(X,Y)`	`B`, when being the responder to an invocation from which `B` can successfully receive `X`, chooses to return `Y` to the invoker.
`B:may.returnFor0(Y)`	`B`, when being the responder to an invocation from which `B` can know that nothing was sent by the invoker, chooses to return `Y`.
refined knowledge predicates	
`B:did.returnFor(X,Y)`	`B` knows that `B` has returned `Y`, in an invocation from which `B` has received `X`.
`B:did.returnFor0(Y)`	`B` knows that `B` has returned `Y` in an invocation from which it could not receive anything.
`A:did.getFromFor(B,X,Y)`	`A` knows that `A` has invoked `B` with input `X`, and that, in the same invocation, `B` returned `Y`.
`A:did.getFromFor0(B,Y)`	`A` knows that `A` has invoked `B` with no input and that, in the same invocation, `B` returned `Y`.

Table 5.8: Refinement relations

Implications for refined behavior predicates		
`B:may.returnFor(X,Y)`	$\Leftarrow$	`B:may.return(Y)`
`B:may.returnFor0(Y)`	$\Leftarrow$	`B:may.return(Y)`
Implications for refined knowledge predicates		
`B:did.returnFor(X,Y)`	$\Rightarrow$	`B:did.receive(X)` $\wedge$ `B:did.return(Y)`
`B:did.returnFor0(Y)`	$\Rightarrow$	`B:did.return(Y)`
`A:did.getFromFor(B,X,Y)`	$\Rightarrow$	`A:did.sendTo(B,X)` $\wedge$ `A:did.getFrom(B,Y)`
`A:did.getFromFor0(B,Y)`	$\Rightarrow$	`A:did.getFrom(B,Y)`

in the invocation. The original behavior is less specific, less conditional, and therefore implies the more specific behavior.

If we know that an entity can sometimes interact, but we cannot specify when and in what circumstances it will interact, we can only safely approximate its behavior by a subject that interacts in all circumstances. But, if we know that the entity only interacts in certain conditions, we should use the refined behavior to model that. The coarsely defined behavior (interact in all circumstances) implies the fine grained behavior (interact in special circumstances).

The new system rules will generate refined knowledge from refined behavior. The refined knowledge is more specific than the original knowledge, just like refined behavior is more specific than the original behavior. The refined knowledge implies the original knowledge for the same reason: it is more specific. If a subject knows it has successfully interacted-in-very-special-conditions with another subject, it also knows that it has successfully interacted to that subject.

5.5.2 Refined Rules

All four system rules defined earlier in section 5.4.3 remain valid and are still included in the refined KBM.

We add four extra system rules to manage the refined behavior. Two new system rules will guard the refinement implications for the behavior predicates (Table 5.8, first part). Two other rules will guarantee the refinement relations for the knowledge predicates (Table 5.8, second part). The former two rules are exceptional, because their postconditions contain behavior predicates instead of knowledge predicates.

Behavior Refinement Rules

To guarantee the refinement relations for the behavior predicates, we introduce a new type of rules: *behavior refinement rules*. These rules will generate more-refined behavior predicates from less-refined behavior predicates to guarantee that every subject's behavior is automatically adapted to the refinement that was introduced.

The behavior refinement rules are identical to the first two rules in table 5.8:

ref1. `S:may.return(Y)` ⇒ `S:may.returnFor(X,Y)`

ref2. `S:may.return(Y)` ⇒ `S:may.returnFor0(Y)`

Refined System Rules

We do not need a new type of rules to specify the refinement relations for knowledge predicates in table 5.8. We enforce these relations by adding two new system rules.

sys5.
```
A:may.sendTo(B,X) ∧ A:may.getFrom(B) ∧ access(A,X)
∧ access(A,B) ∧ B:may.receive()
∧ B:may.returnFor(X,Y) ∧ access(B,Y)
⇒ access(A,Y) ∧ access(B,X) ∧ A:did.getFromFor(B,X,Y)
   ∧ B:did.returnFor(X,Y) ∧ A:did.getFrom(B,Y)
   ∧ B:did.return(Y)
```

sys6.
```
A:may.getFrom(B) ∧ access(A,B) ∧ B:may.returnFor0(Y)
∧ access(B,Y)
⇒ access(A,Y) ∧ A:did.getFrom(B,Y) ∧ B:did.return(Y)
  ∧ A:did.getFromFor0(B,Y) ∧ B:did.returnFor0(Y)
```

Note that the preconditions of rule **sys5** imply the preconditions of rule **sys1** (Section 5.4.3), which already generates the additional knowledge that is required for the refinement relations in table 5.8: `A:did.sendTo(B,X)` and `B:did.receive(X)`.

Subjects interacting by rule **sys5** will not only get knowledge about a successful exchange, but also about the individual parts of the exchange.

This will make sure that subjects who did not refine their behavior using the predicate `may.returnFor` and who did not specify their behavior with behavior rules that depend on `did.returnFor` or `did.getFromFor` knowledge, will not need to change their behavior in the refined KBM. The behavior refinement rules will take care of all necessary adaptations.

The conditions in rule **sys5** state that:

- `A` must be willing to send `X` to `B`
  That is because `B` will decide to return `Y` only after `B` has received `X`.
- `A` must be willing to get access from `B` when invoking `B`
  `A` is the invoker and, since we model collaboration, `A` decides if it wants to accept or reject return values from `B`.
- `A` must have the necessary access permissions
- `B` must be willing to receive access.
  `B` cannot know what subject will be emitted by its invoker and can only decide to return `Y` after it has received `X`.
- `B` must be willing to return `Y` to invokers that send `X` to `B`.

Rule **sys6** is similar to rule **sys2**. It allows the responder to express return behavior that explicitly requires that nothing was emitted to the responder in the invocation context. The specific knowledge generated by that rule will allow the invoker to refine its behavior based on what the responder returns in invocations the invoker did not send anything.

5.5.3 Overloading Knowledge Predicates

Since the preconditions of rule **sys5** imply the preconditions of rule **sys1** (Section 5.4.3), we can try to simplify rule **sys5** as follows:

sys5b.
```
B:may.returnFor(X,Y) ∧ access(B,Y) ∧ A:did.sendTo(B,X)
⇒ access(A,Y) ∧ access(B,X) ∧ A:did.getFromFor(B,X,Y)
  ∧ B:did.returnFor(X,Y) ∧ A:did.getFrom(B,Y)
  ∧ B:did.return(X)
```

Strictly spoken, this is not a valid system rule, because one of its preconditions (underlined) refers to `A`'s knowledge, instead of to a subject's behavior or permissions.

To make the rule valid we have to overload the did.sendTo predicate to represent both a subject's permission and a subject's knowledge.

In practice, rule sys5b can be a valid and efficient representation of rule sys5 in systems that also contain rule sys1, but only when no did.sendTo knowledge is specified in the initial configuration, as a left over knowledge from a previous evolution.

Because of the concise notation of rule sys5b, we will sometimes overload subject knowledge predicates this way, when expressing patterns of collaborating subjects in chapters 7 and 8. We will then always make sure that the overloaded knowledge predicate is not present in the initial configuration.

Using overloading, we can choose to turn the refinement relations for the knowledge predicates in table 5.8, directly into system rules. Some examples in chapter 8 will follow this approach.

5.5.4 A Proof-of-Access Tester with Exchange Behavior

With the refined rules added, the access tester alice from section 5.4.7 (Figure 5.5) can now be expressed more precisely, using the following behavior rules:

```
                    ⇒  carol:may.receive()         (1)
   carol:isPublic(Y)  ⇒  carol:may.return(Y)          (2)
carol:isSecretTkn(X)  ⇒  carol:may.returnFor(X,Y)    (3)
```

Rules (1) and (2) are identical to our first attempt in section 5.4.7 but the effect of rule (2) is different now. The unrefined behavior predicate may.return/2 in (2) indicates that no conditions on the invocation context are specified when returning "public" capabilities. That means that public capabilities will be emitted using may.returnFor0/2 as well as may.returnFor/3, because that is enforced by rules **ref1** and **ref2**. As indicated in (3), the non-public capabilities are only emitted using may.returnFor/3.

Just like the actual access tester it models, the carol subject in our refined KBM can now be invoked with an input argument, check it, return her own secret if the invoker provided the secret token, and afterwards repeat the whole thing over again.

5.5.5 Proxying to an Access Tester

The refined behavior predicates of table 5.7 allow us to extend the invocation context over more that one invocation. Suppose a proxy was interpositioned between the access tester and his clients, and the proxy can be relied upon to forward and return the arguments between the clients and the access tester, just as if it the clients would invoke the access tester directly. Such behavior is easy to implement using a proxy pattern [GHJV94], sometimes referred to as a transparent forwarder pattern. Thanks to the refined predicates, our refined model is expressive enough to model it accurately.

In fact, the code in figures 5.2 and 5.3 implement exactly such a proxy. Our first attempt at modeling that proxy, using the unrefined behavior predicates, was as follows:

```
⇒ proxy:may.receive()                          (1)
proxy:target(T) ∧ proxy:did.receive(X)
⇒ proxy:may.sendTo(T,X)                        (2)
proxy:target(T) ⇒ proxy:may.getFrom(T)         (3)
proxy:target(T) ∧ proxy:did.getFrom(T,X)
⇒ proxy:may.return(X)                          (4)
```

Now suppose that `proxy's` target is a proof-of-access tester, as described in section 5.5.4. This model would be hopelessly inaccurate to express that we rely on `proxy`, not to return the secret values `proxy` returned by its target (in exchange for a secret token provided by one of `proxy`'s clients), indiscriminately to all its clients.

In the refined KBM, we can make sure that `proxy` respects its target's choices about what invoker should get access to the target's secret, by modeling `proxy`'s behavior rules in way:

```
⇒ proxy:may.receive()                               (1)
proxy:target(T) ∧ proxy:did.receive(X)
⇒ proxy:may.sendTo(T,X)                             (2)
proxy:target(T) ⇒ proxy:may.getFrom(T)              (3)
proxy:target(T) ∧ proxy:did.getFromFor0(T,X)
⇒ proxy:may.returnFor0(X)                           (4)
proxy:target(T) ∧ proxy:did.getFromFor(T,X,Y)
⇒ proxy:may.returnFor(X,Y)                          (5)
```

Behavior rules (1) and (2) make sure that our proxy works in the forward way: it will invoke the access tester emitting the arguments it collected as a responder. Behavior rules (3) and (4) make sure that our proxy works in the backward way for those invocations in which nothing is emitted by the invoker. Behavior rules (2),(3) and (5) make sure that our proxy also works in the backward way for those invocations in which something was emitted by the invoker.

Because the access tester will only return its non-public capabilities to invokers that send the secret token in the same invocation, our proxy will receive knowledge about these capabilities only via the `did.getFromFor/4` and `did.getFrom/3` predicates (rule *sys5*). Our proxy does not use its `did.getFrom/3` knowledge, only the more detailed `did.getFromFor0/3` and `did.getFromFor/4` knowledge.

The proxy pattern is a very important and widely used structural design pattern. In [GHJV94] its intent is described as : "*Provide a surrogate or placeholder for another object to control access to it.*". It is crucial that our system can accurately model proxy behavior. Proxies with this refined behavior will play a central role in the patterns for revokable authority and will enable us to calculate a safe and accurate approximation of the conditions in which revokable authority patterns can safely be used(Section 8.2.1).

5.6 More Expressive Power

Until now, we have only scratched the surface of the expressive power that KBMs can provide. Even with the refinement introduced in section 5.5, the model still lacks expressive power for fine grained behavior-based analysis.

In this section we will look into the most important restrictions that still remain and investigate how these restrictions can be overcome to allow software engineers to express more precisely, how the authority propagates in specific parts of their software. We generalize the refinement approach of section 5.5 and propose generic solutions to most of these restrictions.

We do not propose a single one-fits-all solution to enhance the expressive power of KBMs. The generic solutions only show that it is possible, safe, and easy to refine a KBM in a the specific area where more expressive power is needed. These areas will be different for different kinds and different instances of safety problems.

Restricting the refinement efforts to a specific area allows the software engineer to keep a general overview of the problem and concentrate on the parts of the relied-upon code that need refined modeling. The safety problems we will model with KBMs are computationally intensive. Therefore, introducing unnecessary refinement will take more time and computation power to solve the problem.

5.6.1 Restrictions

Multiple arguments : The model has only one input and/or one output argument per invocation. More complex invocations have to be approximated by multiple simple invocations. This restricts the power of the model to express fine-grained behavior.

For instance, responders may want to perform access tests similar to the one in section 5.5.4, on tuples of entities instead of on the entities one by one (proof-of-access to two or more shared secrets).

Modeling data : The current model has no separate way to represent data. We can only model data as a passive subject (no collaborative behavior). Modeling data can be useful to refine the interactions between relied-upon subjects.

Data could for instance be used to express the name of a method that is invoked in an interaction. If we could express that a relied upon subject `p` only invokes certain "safe" methods of a relied upon subject `q`, that would not only reduce the access that `p` can gather by invoking `q` in the model, but indirectly also the access that untrusted subjects can acquire in the model, by invoking `p`.

Context-specific behavior and knowledge: The predicates *may.returnFor()* and *may.returnFor0()* represent behavior in the context of a single response or invocation. It would greatly enhance the expressive power of our formalism, if such context(s) could be made explicitly available to the system rules and the behavior rules.

The developer could then model his relied-upon entities more precisely, by expressing preconditions for their interaction, based on the properties of the invocation context that are visible to the invoked entities.

Non monotonic changes in behavior : Our monotonic approach does not allow us to directly model a subject that changes its behavior in a non-monotonic way (e.g. by using less or completely different behavior rules) when it becomes aware of an event (knowledge).

We have strong reasons to stick to monotonic subject behavior, because it improves the confidence that models are derived safely and because it makes the safety analysis tractable. Within certain limitations, section 5.6.5 will propose a way out of this dilemma.

Language Specific Support: Tools for semi-automated model extraction, e.g. via abstract interpretation, should help programmers to confidently model code, programmed (or to be programmed) in the language of their choice. Such tools can be tailored to cope with the language specific concepts that influence the way in which authority propagates.

For instance, the interaction model that we use in our KBMs is fit to express the propagation of pre-existing capabilities, but may be less suitable to model authority propagation when logic variables are used. A logic variable is a placeholder for a value (or for a whole series of values) and can be propagated by interaction, before becoming bound to an entity or to another logic variable by unification. With logic variables and unification, the flow of authority propagation can always go in both ways: from invoker to responder and back.

While the design or development of language-specific tools for model extraction are out of the scope of this thesis, we consider them to be very interesting future work that will be crucial to attain a generalized accessibility of KBM-based safety analysis.

5.6.2 A Generic Approach to Refinement

We introduce a general strategy to refine knowledge and behavior predicates when needed and we investigate what kind of expressive power our model can gain that way. An appropriate strategy for refining predicates will overcome many of the restrictions mentioned earlier (Section 5.6.1).

We want the existing coarser predicates to keep their current semantics. Adding refined behavior should not influence the meaning of the rules that are expressed without the refined predicates. The old behavior rules should remain valid and stay a safe approximation for the actual behavior they model. Subject rules detect knowledge and generate behavior. More refined (specific) behavior predicates express restricted behavior: willingness to interact in less general conditions. Therefore they are expected to cause less interactions than the corresponding less refined behavior predicates. As a consequence, a behavior predicate should imply all its refinements. We refer to this by the slogan:

Refined behavior is less behavior.

On the other hand, refined knowledge predicates express more specific knowledge and are therefore expected to trigger more behavior rules : both the ones that only require general knowledge and the ones that require specific knowledge. This means that, from the point of view of a subject, a knowledge predicate should always imply all its generalizations. We refer to this by the slogan:

Refined knowledge is more knowledge.

The system rules generate knowledge (interaction effects) from subject behavior and should also remain monotonic. More behavior should lead to more knowledge.

If we want to refine a subject's behavior we can do so by having its behavior rules generate more refined behavior predicates than before from the same knowledge, or by generating the same behavior predicates as before from more refined knowledge predicates. In both cases, the net effect will be: less interaction and less propagation of authority throughout the configuration.

When modeling a proxy subject, we may want to respect and maintain the level of refinement specified in the behavior of the proxy's target. Therefore we must use specialized rules that only depend on refined knowledge and generate refined behavior.

We do not need to adapt the behavior of unknown subjects, modeled for maximal authority propagation. We only need to introduce behavior refinement rules, similar to the ones presented in section 5.5.2, to make sure that the behavior of all subjects respects the intended refinement relation: less refined behavior implies more refined behavior.

Let us express these general ideas formally now. We will represent the refined versions of an arbitrary predicate `p` of arity k using the extended predicate $\texttt{p}'$ of arity $k+1$. The extra element in the extended predicate $\texttt{p}'$ will not always be a subject, but an element of a finite complete lattice $(\mathcal{S}, \leq)$ in which $r \leq s$ indicates that s is equally or more specific than r.

*A **complete lattice** $(P, \leq)$ is a partial order in which all subsets have a greatest lower bound (join) and a least upper bound (meet). The join of two elements `A` and `B` is denoted by $a \sqcup b$. The join of all elements in a set $A \subseteq P$ is denoted by $\bigsqcup A$. The meet of two elements `A` and `B` is denoted by $a \sqcap b$. The meet of all elements in a set $A \subseteq P$ is denoted by $\bigsqcap A$. The top element of the lattice is denoted $\top = \bigsqcup P = \bigsqcap \phi$. The bottom element is denoted $\bot = \bigsqcap P = \bigsqcup \phi$.*
*A **complete join semi-lattice** is a partial order in which all subsets have a greatest lower bound (join)*
*A **join semi-lattice** is a partial order in which all finite subsets have a greatest lower bound (join)*

We induce a join semi-lattice in the extra argument of the extended predicate to express the refinement relations. The original, unrefined predicate $\texttt{p(S}_1\texttt{,}\ldots\texttt{,S}_n\texttt{)}$ will be represented in the extended predicate with the bottom element of the lattice as the extra element: $\texttt{p}'\texttt{(S}_1\texttt{,}\ldots\texttt{,S}_n\texttt{,}\bot\texttt{)}$.

Example

To illustrate the approach, we will express the refinement of `may.return/2`, introduced in section 5.5 using the lattice: $(\{\bot, nil\} \cup \{subjects\}, \leq)$.

The extra element nil will indicate "no subject" and will express the predicates `may.returnFor0/2`, `did.returnFor0/2`, and `did.getFromFor0/2` in table 5.7.

To express the refined predicates `may.returnFor/3`, `did.returnFor/3`, and `did.getFromFor/4` the extra element of the extended predicate will be a subject. The predicates `may.return/2`, `did.return/2`, and `did.getFrom/3` will be expressed using $\bot$ as the extra element in the extended predicate.

In other words, the element nil expresses the requirement that no subject is emitted by the invoker in the same invocation in which an element is emitted by the responder. It differs from $\bot$, which indicates that there is no requirement at all about the elements emitted by the invoker in that invocation. The presence of a subject means that at least one entity was emitted by the invoker, in the invocation.

Formally, our join semi-lattice is $(\mathcal{S}', \leq)$ where :

$\mathcal{S}'$ is the set of subjects $\mathcal{S}$, extended with $\{\bot, nil\}$, and

$\forall x, y \in \mathcal{S}' : x \leq y \Leftrightarrow x = \bot$ or $x = y$.

We can now express the predicates introduced in section 5.5, using the extended predicates shown in table 5.9.

To express the refinement relations between the predicates in terms of this lattice, we extend the specialization partial order from $\mathcal{S}'$ to the set $\mathcal{P}$ of grounded predicates

Table 5.9: Refined behavior using a refinement semi-lattice

predicate	refinement lattice
`S:may.return(Y)`	`S:may.return`$'$`(Y,`$\perp$`)`
`S:may.returnFor(X,Y)`	`S:may.return`$'$`(Y,X)`
`S:may.returnFor0(Y)`	`may.return`$'$`(S,Y,`nil`)`
`S:did.return(Y)`	`S:did.return`$'$`(Y,`$\perp$`)`
`S`$_1$`:did.getFrom(S`$_2$`,Y)`	`S`$_1$`:did.getFrom`$'$`(S`$_2$`,Y,`$\perp$`)`
`S:did.returnFor(X,Y)`	`S:did.return`$'$`(Y,X)`
`S`$_1$`:did.getFromFor(S`$_2$`,X,Y)`	`S`$_1$`:did.return`$'$`(S`$_2$`,Y,X)`
`S:did.returnFor0(Y)`	`S:did.return`$'$`(Y,`nil`)`
`S`$_1$`:did.getFromFor0(S`$_2$`,Y)`	`S`$_1$`:did.getFrom`$'$`(S`$_2$`,Y,`nil`)`

(facts) over $\mathcal{S}$, with labels from a finite set L and arities decided by a function $A : L \rightarrow \mathbb{N}$.

The partial order $(\mathcal{P}, \leq)$, is defined as:

$\mathcal{P} = \{p(x_1, \ldots, x_n) | p \in L, n = A(p), \forall 1 \leq i \leq n : x_n \in \mathcal{S}'\}$

$\forall p(x_1, \ldots, x_n), q(y_1, \ldots, y_m) \in \mathcal{P}$:
$p(x_1, \ldots, x_n) \leq q(y_1, \ldots, y_m) \Leftrightarrow p = q, n = m$, and $\forall 1 \leq i \leq n : x_i \leq y_i$ in $(\mathcal{S}', \leq)$

In the partial order $(\mathcal{P}, \leq)$, all the refinement relations between the predicates shown in table 5.8 can now be expressed generically as is shown in table 5.10. We will adorn variables with a prime sign (e.g. X') to indicate that they range over the extended lattice set, rather than (only) over the set of subjects.

Table 5.10: Refinement relations in the semi-lattice of facts

type	order condition	refinement relation
behavior	$b(X_1, \ldots, X'_n) \leq b(Y_1, \ldots, Y'_n)$	$b(X_1, \ldots, X'_n) \Rightarrow b(Y_1, \ldots, Y'_n)$
knowledge	$k(X_1, \ldots, X'_n) \geq k(Y_1, \ldots, Y'_n)$	$k(X_1, \ldots, X'_n) \Rightarrow k(Y_1, \ldots, Y'_n)$

If $p(X_1, \ldots, X_n) \leq q(Y_1, \ldots, Y_n)$, then $q(Y_1, \ldots, Y_n)$ will cause less interaction if `P` is a behavior predicate and more interaction if `P` is a knowledge predicate.

Remark Notice that $(\mathcal{P}, \leq)$ will most likely not be a semi-lattice. That is OK because we only need a partial order to deduce the refinement relations between the predicates. However, if all predicates would be constructed as refinements, starting from a single predicate $p()$ with arity 0, then $(\mathcal{P}, \leq)$ would be a complete semi-lattice, with $p(\perp, \ldots, \perp)$ as its bottom element.

Generalization of Lattice Based KBM Refinement

Behavior and knowledge predicates, extended for the purpose of refinement, can replace their original form by using the bottom element $\perp$ of a join semi-lattice. This semi-lattice will naturally induce a partial order among the set of extended facts.

The refinement relations between the behavior facts will deduce more specific behavior facts from less specific ones in the induced partial order. The refinement relations between the knowledge facts will deduce more specific knowledge facts from less specific ones in the induced partial order.

The original predicates in the system and behavior rules can, but don't necessarily have to, be replaced by their extended form using the $\perp$ element, as long as its semantics is clear. Alternatively, new labels can be used (e.g `may.returnFor`, `may.returnFor0`) for the refined predicates.

In the examples, we used a join semi-lattice on the set of subjects, extended with $\perp$ and nil, but in general, the semi-lattice can have a completely different nature. In the remainder of this section, we will discuss applications of KBM refinement that make use of different lattices.

Consecutive refinements can use different semi-lattices to express the refinement relations. These semi-lattices can always be combined, by taking their lattice product. The product of a set of lattices is defined as the set of tuples of elements, one of each of the lattices, with the partial order defined per element in the tuple.

$$\begin{aligned}&(P, \leq_P) \times (Q, \leq_Q)\\&= (P \times Q, \leq_{P\times Q}) : (p_1, q_1) \leq_{P\times Q} (p_2, q_2) \Leftrightarrow p_1 \leq_P p_2 \wedge q_1 \leq_Q q_2\end{aligned}$$

The overall effect on the partial order between the facts is defined by this product semi-lattice:

$$p(X'_1, \ldots, X'_n) \leq q(Y'_1, \ldots, Y'_n) \Leftrightarrow \forall 1 \leq i \leq n : X'_i \leq Y'_i$$

The introduction of a semi-lattice that extends the set of subjects in a KBM is only one way to impose a partial order on the predicates. The semi-lattice set does not have to be a superset of the subjects.

Remark When introducing refinements, we have to be careful not to model refined behavior that depends on conditions that cannot be tested by the entities whose behavior we model. The refinements introduced until now were OK, because in our interaction model the number of input arguments in an invocation can be detected by the responder.

5.6.3 Adding data

If we can propagate data together with subjects, we will be able to refine behavior, based on the data that is passed during collaboration. Such data can represent input data arguments and output data arguments. Data can also be used to model the name of a message.

A data refinement semi-lattice can take several forms, depending on what purpose the data has to model. Figure 5.6 shows some examples of such data semi-lattices.

A linear order can be used to indicate the number of arguments (input arguments, output arguments) of a procedure or a method. A flat semi-lattice can also model the name of the message or procedure. A flat semi-lattice was used to refine the $may.return/2$ behavior predicate in section 5.6.2, using subjects, nil and $\perp$, instead of data.

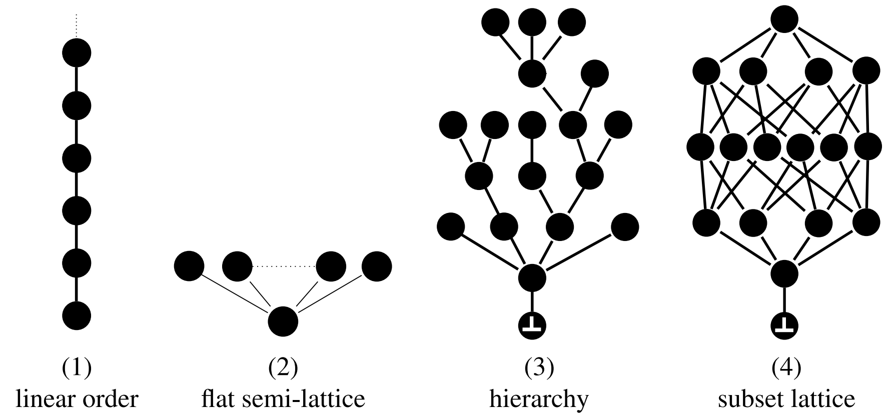


Figure 5.6: Examples of data lattices

A type hierarchy will resemble the structure of the third example in figure 5.6 and can be modeled as an hierarchical semi-lattice. Method signatures in statically typed languages can be completely expressed with appropriate combinations of (2) and (3). In dynamically typed languages a similar type hierarchy can be used to model relied-upon behavior restrictions in the use of types and/or certified (branded) entities.

Sets of independent Boolean (or binary) properties of the message and/or its argument will form a lattice similar to (4).

Note that we added a separate $\perp$ node to the hierarchy and the subset lattices, to draw attention to the fact that the common supertype in (3) and the empty set in (4) do not necessarily coincide with $\perp$. An element representing the supertype (e.g. "Object") may represent the requirement of an argument of any type, while $\perp$ is reserved to represent the absence of a requirement. Likewise, the empty subset can express the requirement of a data argument, with not further requirements about the represented properties.

Another useful and simple lattice is the singleton $\{\perp\}$. It can be used to implement a trivial refinement, which allows us to generalize the presence of a semi-lattice in all KBMs.

5.6.4 Multiple Arguments

To express the propagation of multiple capabilities in a single invocation, we can apply the refinement strategy based on join semi-lattices proposed earlier in this chapter. The behavior predicates can be extended with as many arguments as necessary to express relied-upon restrictions about the extra arguments to be emitted and/or collected in the same invocation.

In combination with the data refinement strategies of section 5.6.3 to model the method name or procedure name, modeling multiple arguments allows us to express restrictions that very precisely match the actual invocations in the code.

5.6.5 Non Monotonic Changes

Instead of directly providing support to model a decrease in an entity's willingness to interact, we propose to model the entity with decreasing behavior as two separate subjects. One subject models the behavior before the change, the other one after the change. The behavior rules in the latter part of the decomposed subject will all include the precondition that models the entity's awareness that the behavior change was reached.

The other subjects do not need to be split up. However, the initial conditions must be carefully adapted to make sure that all subjects have access to both parts of the decomposed subject or to neither of them. In general, no initial conditions should differentiate between both parts of the decomposed subject.

The analysis can then show the difference in authority that will be available to both versions of the subject, reflecting the difference in maximally attainable authority by the subject, before and after its behavior decrease. Most interestingly, the difference in authority that can be provided directly by both versions of the subject, can be derived from the knowledge about successful interaction that became available to both parts of the decomposed subject .

Finding the difference in authority that can be made available indirectly by both versions of the subject will be very interesting future work and can be based on the authority flow analysis that is presented in chapter 9.

5.6.6 Behavior and Knowledge Inheritance

Until now, we only considered a refinement partial order on the extended arguments of the predicates, never on the arguments that indicate a subject whose behavior will be considered or whose knowledge will be influenced by a system rule.

Introducing a partial order between the subjects themselves would allow us to express behavior and/or knowledge inheritance between subjects (instances). That would open possibilities to consider other forms of aggregation, in which :

- Every knowledge *available to* a subject implies that this knowledge is also available to its less specialized versions.
- Every knowledge *about* a subject implies the same knowledge about its less specialized versions.
- Every behavior *of* a subject implies that its specialized versions also have this behavior.
- A subject `carol`'s behavior *regarding* another subject `bob` implies that `carol` also has this behavior regarding `bob`'s specialized versions

Knowledge inheritance in particular, may have interesting applications for non-monotonic behavior. In the approach of section 5.6.5, the subject "lower" in the specialization order could represent the version of the entity after the behavior change. This line of thought will not be further explored in this thesis, but is left as future work.

5.7 Formal Definitions and Proofs

In this section we develop KBMs as a monotonic predicate logic. Safety problems will be expressed as predicates in this logic. The intention is to provide a proof-theoretical base for reachable authority. The logical semantics will also be used to prove that the technique of aggregation results in safe approximations: the safety properties that are guaranteed in an aggregated model are valid for the non-aggregated model.

This result allows us to safely approximate problems with an unbounded number of entities, e.g. created at runtime, using a finite KBM with a fixed set of subjects. A safety property, guaranteed (proven) in this KBM for an aggregated subject `X`, is automatically valid for every entity that was modeled as `X`.

5.7.1 Knowledge Behavior Models

We define Knowledge Behavior Models as tuples that contain all the necessary ingredients to logically derive a behavior-based maximal bound to authority. Remember that this is actually an upper bound to the authority of which we cannot prove that it is not reachable.

These ingredients are:

1. A set of subjects
2. Three disjunct sets of predicate variables to indicate permissions, behavior and knowledge
3. Two finite sets of universally quantified implications, to:
   (a) reflect the possibilities for propagation of authority, offered by the interaction model and allowed by the protection system (the system rules), and
   (b) reflect the monotonically approximated influence of the subjects on the propagation of authority (the behavior rules).

We will use this definition eventually to formally define the safety problems that can be expressed in SCOLL.

Definition 9 (KBM). *Knowledge Behavior Model :*
A KBM is a tuple $\langle S, P_p, P_b, P_k, V, R_s, R_b \rangle$ *such that:*
S *is a set of subjects.*
P_p, P_b *and* P_k *are disjunct sets* $\{p, q, \ldots\}$ *of predicate variables, each with finite arity* $ar(p) > 0$.
V *is an enumerable set of variables, each ranging over the set of subjects.*
R_s *is a sets of universally quantified formulas called* `rules` *of the form:*

$$\forall X, Y \ldots : p_1(X_{1,1}, \ldots, X_{1,ar(p_1)}) \wedge \ldots \wedge p_n(X_{1,n}, \ldots, X_{n,ar(p_n)}) \\ \rightarrow q(X_{n+1,1}, \ldots, X_{n+1,ar(q)})$$

with $n \geq 0$,
$p_i \in P_p \cup P_b$ *and* $q \in P_p \cup P_k$
$X_{i,j} \in V$
and all variables are universally quantified.

R_b *is a sets of universally quantified formulas called* `rules` *of the form:*

$$\forall X, Y \ldots : p_1(X_{1,1}, \ldots, X_{1,ar(p_1)}) \wedge \ldots \wedge p_n(X_{1,n}, \ldots, X_{n,ar(p_n)}) \\ \rightarrow q(X_{n+1,1}, \ldots, X_{n+1,ar(q)})$$

with $n \geq 0$,
$p_i \in P_k$ *and* $q \in P_b \cup P_k$
$X_{i,j} \in V \cup S$ *with* $X_{i,j} \in S \Leftrightarrow i = 1$ *and* $X_{1,j} = X_{1,j+1}$
and all variables are universally quantified.

A KBM is finite *if* S, P_p, P_b, P_k, R_s, *and* R_b *are finite.*

Notations

The distinction between permissions, behavior, and knowledge is crucial to our approach when modeling software. However, for our logical semantics these distinctions are irrelevant. We will therefore usually denote a KBM in its shorter form as a tuple $\langle S, P, V, R\rangle$ such that: $P = P_p \cup P_b \cup P_k$ and $K = K_s \cup K_b$.

Outside the context of a KBM $G = \langle S, P, V, R\rangle$, we will denote S as S_G, P as P_G, V as V_G, and R as R_G.

Definition 10. *The logical language of a KBM*
A KBM G = $\langle S, P, V, R\rangle$ defines a logical language L_G to reason about the subjects in S:

- *the **terms** of L_G are symbols that uniquely name:*
  - *the subjects in S*
  - *the variables in V*
- *the **well formed formulas** of L_G are recursively defined as:*
  - *atomic formulas: $p(t_1, \ldots, t_{ar(p)})$ where $p \in P$ and all t_i are terms in L_G.*
  - *negations: $\neg F$, where F is a well formed formula.*
  - *conjunctions: $F_1 \wedge F_2$, where F_1 and F_2 are well formed formulas.*
  - *disjunctions: $F_1 \vee F_2$, where F_1 and F_2 are well formed formulas.*
  - *implications: $F_1 \rightarrow F_2$, where F_1 and F_2 are well formed formulas.*
  - *universal quantifications: $\forall X_1, X_2, \ldots : F$, where $X_i \in V$ and F is a well formed formula.*
  - *existential quantifications: $\exists X_1, X_2, \ldots : F$, where $X_i \in V$ and F is a well formed formula.*
  - *nothing else.*

Clearly the rules in R all correspond to well formed formulas. When convenient, we will reuse R to denote this set of formulas. All formulas in R are closed: because of the universal quantification, no variable is free in any of the formulas. This means that they are sentences in L_G.

A logical language will allow us to prove or disprove sentences, given a set of sentences of which the truth is stated (a theory). Our theories will include the system rules and behavior rules in R , but also a set of ground atomic formulas (predicates with no variable terms) to describe the initial conditions.

Definition 11. *KBM Configuration*
Let G = $\langle S, P, V, R\rangle$ be a KBM.
A configuration of G is an enumerable set of ground atoms in L_G.

Definition 12. *Extension*
Let G = $\langle S, P, V, R\rangle$ be a KBM, and let C be a configuration of G.
The extension of C is the unique assignment of actual predicates to the predicate variables $p \in P$, denoted $ext(C) = \{p_C | p \in P\}$ such that:
$p_C : S^{ar(p)} \rightarrow \{true, false\}$
with $p_C(s_1, \ldots, s_{ar(p)}) \Leftrightarrow (R \cup C) \vdash p(s_1, \ldots, s_{ar(p)})$

In definition 12 we used the symbol $\vdash$ that indicates deductive-theoretic consequence [McK]. This type of logical consequence indicates that $p(s_1, \ldots, s_{ar(p)})$ can be proven in a finite number of deduction steps from the theory $R \cup C$.

This definition needs some explanation. If we had used model-theoretic logical consequence (indicated by the symbol $\models$) instead, our definition would have been more general: it would then state that every model for $R \cup C$ is also a model for $p(s_1, \ldots, s_{ar(p)})$. This type of logical consequence is indeed the most fundamental one: it defines the truth of a sentence, irrespective of its provability. The arguments for this appreciation are given in "The Internet Encyclopedia of Philosophy" [McK], based on Tarski's "The Concept of Logical Consequence" [Tar83].

The rationale for using deductive-theoretic consequence is in the Curry-Howard isomorphism, which defines the equivalence between logical proofs (in intuitionistic logic) and programs. Because the reachability of all states that are reachable in a program is proven by the corresponding execution of a program, we can restrict our attention to the states whose reachability is provable by means of a finite execution of the program. It follows from the Curry-Howard isomorphism that the reachable states correspond to provable predicates in intuitionistic logic.

5.7.2 Proving and Disproving Sentences

We want to mechanically prove or disprove the safety properties and liveness possibilities in L_G, in a time polynomial to the size of our problem. Because a proof is a finite series of valid deductions from the "theory" that corresponds to a configuration and from previous deductions, we have to :

1. define what the valid deductions are
2. and find an algorithm that generates a proof in polynomial time.

We will use the common deduction rules of first order predicate logic, called the elimination and introduction rules for $\neg$, $\wedge$, $\vee$, $\perp$ $(false)$, $\top$ $(true)$, $\rightarrow$, $\exists$, and $\forall$ (restricted to the intuitionistic subset which has not introduction of $\neg$). All these are simple, mechanically applicable rules that can be implemented in software.

In finite KBMs, all predicates $p(s_1, \ldots, s_{ar(p)})$ that can be proven from the theory corresponding to a configuration C of the KBM $\langle S, P, V, R \rangle$, can be found in polynomial time. That will of course allow us to find the remaining, non-provable predicates, in polynomial time too.

Because $R \cup C$ can never be inconsistent (no negation is allowed in R or C) it follows that:

$$(R \cup C) \nvdash p(s_1, \ldots, s_{ar(p)}) \Leftrightarrow (R \cup C) \vdash \neg p(s_1, \ldots, s_{ar(p)})$$

Infinite KBMs will be safely approximated by finite ones.

Because the sentences in our theory only contain universal quantifications, conjunctions and implications (no negations), an implementation of KBMs in the Datalog programming language [GM78] would be straight forward. However, in chapter 7, we will discuss a prototype implementation based on a constraint programming (CP) library [Sch02] which offers many possibilities to optimize search strategies. This has the additional advantage that future extensions will not necessary remain confined to monotonic logic.

Definition 13. ***Safety properties and liveness possibilities***
Let G = $\langle S, P, V, R\rangle$ *be a KBM and let C be a configuration of G*
The set of ***liveness possibilities*** *for C is the set :*

$$Live_G(C) = \{p(s_1, \ldots, s_{ar(p)}) | p \in P, s_i \in S, (R \cup C) \vdash p(s_1, \ldots, s_{ar(p)})\}.$$

The set of ***safety properties*** *for C is the set :*

$$Safe_G(C) = \{p(s_1, \ldots, s_{ar(p)}) | p \in P, s_i \in S, (R \cup C) \nvdash p(s_1, \ldots, s_{ar(p)})\}.$$

These sets correspond to the extension of C:

$$Live_G(C) = \{p(s_1, \ldots, s_{ar(p)}) | p_C \in ext(C), p_C(s_1, \ldots, s_{ar(p)}) = true\}.$$
$$Safe_G(C) = \{p(s_1, \ldots, s_{ar(p)}) | p_C \in ext(C), p_C(s_1, \ldots, s_{ar(p)}) = false\}.$$

Note: A KBM will be used to safely approximate a concrete safety problem: what is impossible in the KBM is impossible in the concrete problem (safety property), but what is not impossible is certainly not guaranteed. Therefore we explicitly avoid the term *liveness properties* and use *liveness possibilities* instead.

5.7.3 Formal Notion of Safe Approximation

To formalize the concept of safe approximation, we will assume that the actually reachable authority in software is defined by a theory K in L_G, where $G = \langle S, P, V, R\rangle$ is a KBM. That makes sense because:

- the operational semantics of the program can be expressed as a state machine.
- the authority in the software can be expressed as predicates about the (history of the) state.
- the state transitions can be modeled as implications in L_G :
  $$\bigwedge\{preconditions\} \rightarrow \bigwedge\{postconditions\}$$
  where the preconditions and postconditions are expressed using predicates about the (history of the) state.

Figure 5.7 shows a sketch of this conceptual transformation. It will usually not be practical to consider the program's actual operational semantics. A more realistic approach would use abstract interpretation of the source code to construct a simplified state machine that safely approximates the reachable states. Authority can then be defined in terms of predicates over the simplified states.

Such an approach would be closer to the intuitive transformation process, presented in the previous sections. Techniques for transforming to and from source code will be discussed as future work in chapter 11.

Definition 14. ***Safe Approximation***
Let G = $\langle S, P, V, R\rangle$ *be a KBM.*
Let C be a configuration G.
C a safe approximation of a theory K in $L_G \Leftrightarrow$

$$\forall p \in P : K \vdash p(s_1, \ldots, s_{ar(p)}) \Rightarrow (R \cup C) \vdash p(s_1, \ldots, s_{ar(p)})$$

G is a safe approximation of a theory K in L_G *if all its configurations C are a safe approximation of* $K \cup C$.

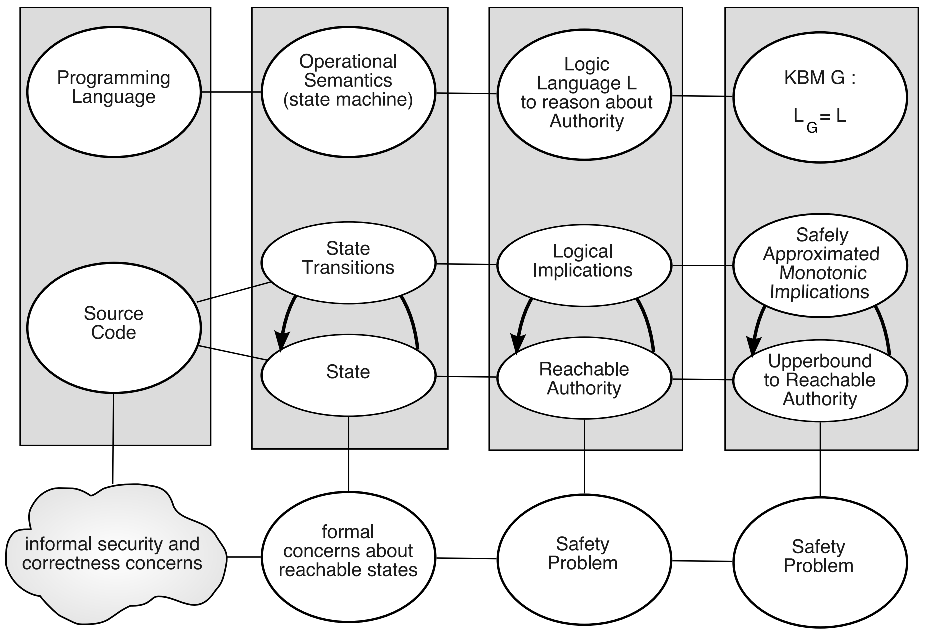

Programming Language
Operational Semantics (state machine)
Logic Language L to reason about Authority
KBM G : L_G = L
Source Code
State Transitions
State
Logical Implications
Reachable Authority
Safely Approximated Monotonic Implications
Upperbound to Reachable Authority
informal security and correctness concerns
formal concerns about reachable states
Safety Problem
Safety Problem

Figure 5.7: Safe approximation via operational semantics

5.7.4 Formal Notion of Aggregation

Aggregation is defined as a partition on the set of subjects in a KBM.

Definition 15. ***Aggregation***
Let $G = \langle S, P, V, R\rangle$ be a KBM and let C be a configuration of G.
Let $A : S \to S$ be an idempotent operation (defining a partition)
Denote $A(S) = \{A(s)|s \in S\}$
Reuse A to denote a transformation operation in L_G via its constant terms:
Let $A(X)$ denote X if $X \in V$
Let $A(p(X, \dots, X_{ar(p)}))$ denote $p(A(X_1), \dots, A(X_{ar(p)})))$
Extend A likewise to all formulas in L_G.
Denote $A(R) = \{A(r)|r \in R\}$.
Denote $A(C) = \{A(p(s, \dots, s_{ar(p)}))|p(s, \dots, s_{ar(p)}) \in C\}$.
A is an aggregation *from G to $A(G) = \langle A(S), P, V, A(R)\rangle$.*

It is easily checked that $\langle A(S), P, V, A(R)\rangle$ is a KBM and that all formulas in $L_{A(G)}$ are formulas in L_G. That makes it easier to express the relations between the safety properties in both KBMs, because we can express everything in the same language.

Theorem 1. ***Aggregation respects Safe Approximation***
Let $G = \langle S, P, V, R\rangle$ be a KBM
Let C be a configuration of G
Let $A : S \to S$ be an idempotent operation.
Let $G' = \langle A(S), P, V, A(R)\rangle$ be the aggregation of G by A.
For every theory K in L_G such that C is a safe approximation to K:
if $(C \cup K) \vdash p(s_1, \dots, s_{ar(p)})$ then also: $A(C \cup R) \vdash A(p(s_1, \dots, s_{ar(p)}))$.

Proof. We observe that for every deduction rule in our first order predicate logic, with n preconditions $E_i : 0 \leq i \leq n$ and postcondition F:

if a rule has the form : $\dfrac{E_1, \ldots, E_n}{F}$

then : $\dfrac{A(E_1), \ldots, A(E_n)}{A(F)}$ is an instance of that rule.

To illustrate this, an example for the modus ponens rule can suffice:

from $A(E) \rightarrow A(F)$, via *modus ponens*: $\dfrac{E \rightarrow F, E}{F}$

we can deduce: $A(F)$

This is obvious if F is an atomic formula and for the other formulas it follows directly from the recursive definition of A.

From definition 15, the correspondence between the theories is defined by the configurations and the rules in G and in $A(G)$, which means that :

$E \in (R \cup C) \Rightarrow A(E) \in A(R \cup C)$.

Because a proof in L_G is a finite series of applications of deductions rules it follows that every deduction rule applied in the proof for $p(s_1, \ldots, s_{ar(p)})$ in L_G can be applied in the proof for $A(p(s_1, \ldots, s_{ar(p)}))$ in $L_{A(G)}$:

if $(C \cup R) \vdash p(s_1, \ldots, s_{ar(p)})$ then also: $A(C \cup R) \vdash A(p(s_1, \ldots, s_{ar(p)}))$.

and herefore :

if $(C \cup K) \vdash p(s_1, \ldots, s_{ar(p)})$ then also: $A(C \cup R) \vdash A(p(s_1, \ldots, s_{ar(p)}))$.

□

Corollary 1. *Safety properties of aggregations are valid*
Let $G = \langle S, P, V, R \rangle$ *be a KBM.*
Let C *be a configuration of* G*.*
Let $A : S \rightarrow S$ *be an aggregation from* G *to* $A(G)$*.*
then: $A^{-1}(Safe_{A(G)}(A(C))) \subseteq Safe_G(C)$

Proof. From theorem 1 :
if $(C \cup R) \vdash p(s_1, \ldots, s_{ar(p)})$ then also: $A(C \cup R) \vdash A(p(s_1, \ldots, s_{ar(p)}))$
thus: if $A(C \cup R) \nvdash A(p(s_1, \ldots, s_{ar(p)}))$ then also: $(C \cup R) \nvdash p(s_1, \ldots, s_{ar(p)})$
thus: if $A(C \cup R) \vdash A(\neg p(s_1, \ldots, s_{ar(p)}))$ then also: $(C \cup R) \vdash \neg p(s_1, \ldots, s_{ar(p)})$
thus: $A(p(s_1, \ldots, s_{ar(p)})) \in Safe_{A(G)}(A(C)) \Rightarrow p(s_1, \ldots, s_{ar(p)}) \in Safe_G(C)$
thus: $p(s_1, \ldots, s_{ar(p)}) \in A^{-1}\left(Safe_{A(G)}(A(C))\right) \Rightarrow p(s_1, \ldots, s_{ar(p)}) \in Safe_G(C)$

□

Corollary 2. *Finite Aggregations are Safe and Tractable Approximations*

Proof. Follows directly from the finite chararcter of the KBM and from theorem 1. □

This corollary means that by :

- modeling a (possibly infinite) set of entities (subjects) onto a finite set of subjects ($A(S)$),
- taking care that all behavior of all the original entities (subjects) is accounted for in the aggregate subject (($A(R)$) and

- taking care that the initial configuration is reflects the all initial access and knowledge in the aggregated subjects $((A(C))$,

the set of safety properties in the aggregated KBM can be calculated in polynomial time and represent a (possibly infinite) set of safety properties in the original system that are all guaranteed!

5.7.5 Expressing Safety Problems with KBMs

Definition 16. ***Safety Problem***
Let G = $\langle S, P, V, R\rangle$ *be a KBM. Let C be a configuration of G.*
Let $X \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of required safety properties.*
The ***safety problem*** *is to decide if* $X \subseteq Safe_G(C)$*.*

Definition 17. ***Practical Safety Problem***
Let G = $\langle S, P, V, R\rangle$ *be a KBM. Let C be a configuration of G.*
Let $X \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of required safety properties.*
Let $Y \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of required liveness possibilities.*
The ***practical safety problem*** *is to decide if* $Y \subseteq Live_G(C) \wedge X \subseteq Safe_G(C)$*.*

We call this variant *practical*, because its detects whether the model guarantees the required safety properties, while not also at the same time preventing required liveness possibilities.

Definition 18. ***Behavior Maximization Problem***
Let G = $\langle S, P, V, R\rangle$ *be a KBM. Let C be a configuration of G.*
Let $U \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of optional behavior.*
Let $X \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of required safety properties.*
Let $Y \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of required liveness possibilities.*
The ***behavior maximization problem*** *is to find a maximal set* $V \subseteq U$:
$Y \subseteq Live_G(C \cup U) \wedge X \subseteq Safe_G(C \cup U)$*.*

Behavior maximization problems are the category of problems we are most interested in in this thesis: to find in a pattern of collaborating entities, the combinations of maximally collaborative behavior for a subset of relied-upon entities that guarantee a safety policy while not preventing the functionality the pattern is supposed to provide. What behavior can we maximally enable our relied-upon entities with, for the pattern to be useful and secure.

Definition 19. ***Knowledge Maximization Problem***
Let G = $\langle S, P, V, R\rangle$ *be a KBM. Let C be a configuration of G.*
Let $U \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of optional behavior.*
Let $X \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of required safety properties.*
Let $Y \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of required liveness possibilities.*
The ***knowledge maximization problem*** *is to find a maximal set* $V \subseteq U$:
$Y \subseteq Live_G(C \cup U) \wedge X \subseteq Safe_G(C \cup U)$*.*

Knowledge maximization problems can be used to look for maximal sets of initial permissions(knowledge facts) that make a pattern useful and secure, when the behavior of the relied-upon subjects in the patterns cannot be (re-)configured.

Definition 20. ***Configuration Maximization Problem***
Let $G = \langle S, P, V, R\rangle$ *be a KBM. Let* C *be a configuration of* G*.*
Let $U \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of optional facts (both knowledge and behavior).*
Let $X \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of required safety properties.*
Let $Y \subseteq \{p(s_1, \ldots, s_{ar(p)}|p \in P$ *and* $s_i \in S\}$ *be called a set of required liveness possibilities.*
The ***configuration maximization problem*** *is to find a maximal set* $V \subseteq U$:
$Y \subseteq Live_G(C \cup U) \wedge X \subseteq Safe_G(C \cup U)$.

Configuration maximization problems are a generalization of behavior and knowledge maximization problems.

Chapter 6

The Language SCOLL

This chapter describes the declarative language SCOLL: Safe Collaboration Language. Many concepts and terms that have been defined and explored in chapter 5 will be used in this chapter, without further explanation.

SCOLL is the language we will use to express patterns of interacting subjects and the safety problems we want to solve considering such patterns. It is based on a kernel language of which we will express the semantics in terms of the Knowledge Behavior Models (KBMs) of chapter 5. The semantics of the full language can then be expressed in terms of the kernel language.

The language is purely declarative and does not allow the user to express how the problem that is described should be solved. That is left to the implementation of the language and will be discussed in chapter 7.

The language SCOLL can be extended by means of linguistic abstraction and syntactic sugar. These techniques are well established practices in programming language design and are described by Van Roy and Haridi in "*Concepts, Techniques, and Models for Computer Programming*"[VH04].

6.1 Objectives

Like the formalism presented in the previous chapter, the SCOLL language has a practical purpose: to help software designers and developers to understand the required behavior- and permission-restrictions for (sets of) programmed entities, given a global safety goal and a context (pattern) in which these entities will play a well-defined role.

The language must therefore not only be simple and expressive, but also provide support for the specific needs of its intended users, the developers of secure software:

1. The structure of the language should allow developers to describe the different aspects of the problem separately, to encourage the reuse of common parts and the exploration of the effects of varying every part independently from the other parts.

2. The interdependencies between the different parts should be simple, unambiguous, and completely described.

3. When expressing a programmed entity's behavior restrictions in SCOLL, the most important errors are the ones that compromise the validity of the model as a

safe approximation of the effects of interaction between the entities. Therefore, the easiest way to approximate a programmed entity's behavior should be the safest way.

4. Improving the accuracy of the modeled behavior should never lead to an underestimation of eventual authority.

5. The language should allow the user to refine a subject's behavior, without having to adapt the behavior of the other subjects. It should encourage an iterative and incremental approach to behavior refinement, starting from crude safe approximations and refining behavior predicates only when the results fail to indicate that a pattern is safe.[1]

6. The language should provide flexible support for aggregation. Aggregation is a technique, expained in section 5.3.1 and formally defined in section 5.7.4, that allows (possibly infinite) sets of programmed entities (including entities that do not exist in the initial configuration but may be created later) to be modeled as a single subject. Theorem 1 in section 5.7.4 proved that aggregation provides valid results for the different problems that can be expressed using KBMs.

6.2 Structure of a Kernel SCOLL Program

In the kernel language, a SCOLL program has a simple structure that roughly corresponds to the parts that constitute a model of interacting entities in section 5.3, with additional parts to describe the initial configuration and the requirements we want to impose:

1. Declarations : Declaration of the predicates with their arity.

2. System : System rules to express the preconditions for successful collaboration and its effects.

3. Behavior : Behavior declarations to describe the types of behavior present in the pattern.

4. Subject : Declaration of the subjects in the pattern.

5. Configuration : The initial permissions and authority.

6. Goal : The policy to be respected.

Throughout a SCOLL Program, several types of identifiers will be used. Their syntax is represented in table 6.1 and will be expressed formally in figure 6.3 (Section 6.3).

We now describe every part separately. The examples provided in this section are only intended to clarify the structure of every part. Together they describe a SCOLL program, depicted in figure 6.1, that is valid in the kernel language, but not necessarily meaningful or realistic. Section 6.9 will provide a practical example, expressed in the complete SCOLL language.

[1]SCOLL is designed to detect when safety is guaranteed, not to detect when safety is broken. A failure to detect that safety is guaranteed may indicate that the relied-upon behavior in the pattern was not expressed to the appropriate level of precision.

```
declare
   permission :  access/2
   behavior :    may.grantTo/3
   knowledge :   did.grantTo/3 did.receive/2
system
   access(A,B) access(A,X) A:may.grantTo(B,X)
    => access(B,X)
   access(A,B) access(A,X) A:may.grantTo(B,X)
    => A:did.grantTo(B,X)
   access(A,B) access(A,X) A:may.grantTo(B,X)
    => B:did.receive(X)
behavior
   FORWARDER {
      did.receive(X) forwardTo(P) => may.grantTo(P,X)
      isSelf(Fwdr) => may.grantTo(X,Fwdr)
      }
   UNRESTRICTED {
      => may.grantTo(X,Y)
      }
   PASSIVE {}
subject
      f1:     FORWARDER
      f2:     FORWARDER
   ?  alice:  PASSIVE
      bob:    UNRESTRICTED
      carol:  PASSIVE
config
      access(f1,f1) access(f1,f2) f1:isSelf(f1)
      f1:isProxy(f2) access(f2,f2) access(f2,bob)
      access(f2,carol) f2:isSelf(f2) f2:isProxy(carol)
   ?  f2:isProxy(bob)
      access(alice,alice) access(alice,f1)
      access(bob,bob)
      access(carol,carol)
goal
      access(bob,carol)
   !  access(bob,alice)
```

Figure 6.1: An example of a complete SCOLL program

Table 6.1: Identifiers in SCOLL

type of identifier	syntax	example
variable	`[A-Z][0-9,a-z,A-Z]`*	`A1b`
subject	`[a-z][0-9,a-z,A-Z]`*	`alice01`
behavior	`[A-Z]`$^+$	`FORWARDER`
predicate label	`[a-z][0-9,a-z,A-Z,\.]`*	`may.sendTo`

6.2.1 Declarations

This part consists of the reserved word `declare`, followed by:
“`permission :`”,
a sequence of predicate declarations for permissions,
“`behavior :`”,
a sequence of predicate declarations for behavior,
“`knowledge :`”, and
a sequence of predicate declarations for non-private subject knowledge

```
declare
   permission :  access/2
   behavior :    may.grantTo/3
   knowledge :   did.grantTo/3 did.receive/2
```

Figure 6.2: Example `declare` part

A predicate declaration consists of a unique predicate label identifier, followed by a “`/`” and a strict positive integer to indicate the arity of the predicate.

SCOLL distinguishes four kinds of predicates, of which the first three must be declared in the declare part:

Permission predicates are also called **system knowledge predicates**. They are declared after the `permission` keyword and express the permissions in the modeled system. They represent properties of subjects and relations between them that are known and managed by the system. They are used as prerequisites for authority propagation and also to express the effects of authority propagation. Permissions are not visible to the subjects holding them, at least not directly. Section 6.8.1 will explain why.

Behavior predicates are declared after the `behavior` keyword. They model the aspects of an entity’s programmed behavior that can have an influence on the propagation of authority. The first argument in the predicate indicates the subject whose behavior the predicate refers to. Most examples will adopt the convention of labeling behavior predicates with a verb stem preceded by the `may.` prefix (Section 5.3.2).

Subject knowledge predicates are declared after the `knowledge` keyword and express a subject’s knowledge of a successful interaction. They represent the part of the effects of authority propagation that is visible to the subject. Most examples will adopt the convention of labeling these predicates with a verb stem preceded by the `did.` prefix (Section 5.3.2). The first argument in the predicate indicates the subject that has the knowledge expressed by the predicate.

Private knowledge predicates are not declared here. They will be automatically declared in the *behavior* part (Section 6.2.3), upon their first use in the description of a behavior type.

When appropriate, we will use the alternative notation for behavior and knowledge predicates that was introduced in section 5.3.2. We position the first argument in front of the label, followed by a colon. Instead of the normal notation `b(S`$_1$`,...,S`$_n$`)`, we write: `S`$_1$`:b(S`$_2$`,...,S`$_n$`)`.

In some parts, predicates will be expressed over variables, while in other parts all arguments will be subjects (constants). Predicates over subjects are called **facts**. Contrary to our notation in section 5.3.4, SCOLL predicates never mix subject constants with variables.

We will refer to permission predicates, subject knowledge predicates, and private knowledge predicates with the general term "knowledge predicates". Likewise, permission facts, subject facts, and private knowledge facts will be indicated as "knowledge facts".

In the `behavior` part (Section 6.2.3), all predicates will be used with one less argument than the arity they are declared with: the first argument is implicit and therefore dropped. Section 6.2.3 will explain why.

The example in figure 6.2 declares a single permission predicate `access` with arity `2`, a behavior predicate `may.grantTo` with arity `3`, and the subject knowledge predicates `did.grantTo` and `did.receive` with arities `3` and `2` respectively.

6.2.2 System

This part consists of the reserved word `system`, followed by a sequence of Horn clauses that express how new knowledge facts will be derived from existing knowledge facts and behavior facts. The Horn clauses in this part are also referred to as **system rules**. They consist of a body, the reserved word "=>" to indicate an implication, and a head.

```
system
    access(A,B) access(A,X) A:may.grantTo(B,X)
     => access(B,X)
    access(A,B) access(A,X) A:may.grantTo(B,X)
     => A:did.grant(B,X)
    access(A,B) access(A,X) A:may.grantTo(B,X)
     => B:did.receive(X)
```

Figure 6.3: Example `system` part

The body of a system rule consists of a sequence of predicates over subject variables, representing a conjunction of the conditions expressed by the individual predicates. The head of the Horn clause consists of a single predicate over variables.

All occurrences of a predicate must have been declared in the `declare` part and be used with the number of arguments indicated by its arity. All subject variables are implicitly declared upon the first use of their identifier in a rule and have a scope that coincides with the rule.

In the kernel language, only permission predicates and behavior predicates can be present in the body of a system rule. Only a permission predicate or a subject knowledge predicate can form the head of a system rule.

The example in figure 6.3 expresses three system rules with the same condition. The complete language will allow multi-headed Horn clauses to express such a set of rules concisely.

The first rule expresses the propagation of an `access()` permission (to a subject referenced by variable `X`) from a subject referenced by variable `A` to a subject referenced by variable `B`. For such propagation to happen, `A` must have prior `access()` to `B` and to `X`, and `A`'s behavior must indicate its intention to grant this access permission to `B`.

The other two rules express what knowledge will become available to subjects `A` and `B`, respectively `A:did.grantTo(B,X)` and `B:received(X)`.

This is an example of a non-collaborative system (Section 1.3.8). Collaborative systems have at least one system rule with preconditions on the behavior of more than one subject involved in the propagation. Because of this, collaborative systems provide ways to restrict authority propagation that rely upon the behavior restrictions of subjects that are strategically inter-positioned between the untrusted subjects in the pattern.

6.2.3 Behavior

This part consists of the reserved word `behavior`, followed by a sequence of behavior declarations. Every behavior declaration consists of a unique behavior identifier, followed by "{", a sequence of behavior rules, and "}".

```
behavior
    FORWARDER {
        did.receive(X) forwardTo(P) => may.grantTo(P,X)
        isSelf(Fwdr) => may.grantTo(X,Fwdr)
        }
    UNRESTRICTED {
        => may.grantTo(X,Y)
        }
    PASSIVE {}
```

Figure 6.4: Example `behavior` part

Every behavior rule is a Horn clause, consisting of a body, "=>", and a head. The body consists only of subject knowledge predicates, both public (declared in the `declare` part) and private. The head is a behavior predicate, declared earlier in the `declare` part.

The private knowledge predicates are declared upon their first use in the body of a behavior rule. Their scope stretches over all rules in the behavior declaration, indicated by the enclosing { and }.

Like in system rules, only subject variables (no constants) are used in the predicates. All subject variables are implicitly declared upon the first use of their identifier in a rule and have a scope that coincides with the rule.

All predicates in this part are represented with an implicit (not shown) first argument variable, which makes it look as if the predicates are used with one less argument. The private knowledge predicates that are declared in this section will therefore have an arity of 1 + the number of arguments used in their first occurrence. Upon instantiation of a rule, when the variables are substituted by subjects in the implementation of the language, the implicit first argument will be replaced by the actual subject whose behavior the instantiated rule expresses.

Making the first argument implicit prevents the developer from reusing that argument's variable identifier in the other parts of the rule to refer to "self" and thereby express implicit conditions. We keep all conditions explicit to prevent the developer from making modeling mistakes when expressing behavior. If the developer forgets or drops a condition in a behavior rule, the behavior is still safely approximated, because the rule will only be triggered in more circumstances.

Where relevant, self referencing can be modeled explicitly with private knowledge predicates like `isSelf(X)`, as is shown in the `FORWARDER` behavior of the example in figure 6.4.

This example declares three types of behavior. The underlined first occurrences of the predicates `forwardTo()` and `isSelf()` are both declared with arity 2, within the scope of all behavior rules of `FORWARDER` behavior. All other predicates were declared in the `declare` part. All predicates have an implicit (invisible) variable in the first argument.

The `FORWARDER` type expresses `may.grantTo()` behavior on condition of having received access to the subject (`X`) it will pass on and having `forwardTo()` knowledge (private) about the subject (`P`) it will grant access-to-`X` to. Its second rule indicates that it also wants to grant access to the subjects it knows to be `isSelf()` to all subjects.

The `UNRESTRICTED` type expresses unconditional `grant()` behavior.

The `PASSIVE` type expresses no behavior, in no circumstances, simply by using an empty sequence of rules.

At this point the reader may wonder why the behavior rules express behavior, rather than behavior restrictions. It may seem awkward or wrong, from a certain point of view, that expressing the unrestricted behavior of unknown entities (e.g. `UNRESTRICTED`) requires more work than expressing uncooperative behavior (e.g. `PASSIVE`). Section 6.5.4 will introduce safe defaults for behavior. Section 6.7.7 will explain why behavior restrictions should not be modeled directly. Section 6.8.9 will show how programmed behavior should be mapped to subject behavior in SCOLL.

6.2.4 Subject

This part consists of the reserved word `subject`, followed by a sequence of subject declarations. A subject declaration consists of an optional question mark "?", a subject identifier, a colon ":", and a behavior identifier.

```
subject
          f1:     FORWARDER
          f2:     FORWARDER
      ?   alice:  PASSIVE
          bob:    UNRESTRICTED
          carol:  PASSIVE
```

Figure 6.5: Example `subject` part

All subjects are declared in this part. Their behavior is indicated by the behavior identifier in their declaration and must have been declared in the `behavior` part of the SCOLL program.

The optional "?" indicator marks a subject to be maximized for behavior. If any subject is marked with "?", the SCOLL program expresses a behavior maximization problem (Definition 18, Section 5.7.2). The behavior of all marked subjects will be maximized.

The solution to a behavior maximization problem is a list of zero or more maximal sets of behavior facts. Each set will contain the behavior facts for zero or more "?"-marked subjects, that can be added to its behavior unconditionally, without violating the required safety properties and without preventing the required liveness possibilities. The sets in the solution are maximal, in the sense that adding one more behavior fact to such a set will no longer guarantee the safety properties of the SCOLL program. The safety properties and liveness possibilities will be expressed in the `goal` part (Section 6.2.6).

Five subjects are instantiated in the example of figure 6.5. Two of them, `f1` and `f2`, have `FORWARDER` behavior. The behavior of `bob` is `UNRESTRICTED`, while `carol` is `PASSIVE`. The behavior of `alice` will be maximized, starting from `PASSIVE`.

6.2.5 Configuration

This part consists of the reserved word `config`, followed by a sequence of knowledge facts, each of them optionally preceded by a question mark "?". Knowledge facts are grounded instances of knowledge predicates, including permissions and private knowledge predicates. All facts are represented with explicit (visible) first argument. The arguments in the facts must be declared in the *subject* part.

All facts have to be derived from knowledge predicates that were defined in one of the previous parts of the SCOLL program. Private knowledge facts must be instances of a private knowledge predicate that was declared in the behavior declaration of the subject in the first argument of the fact.

```
config
       access(f1,f1) access(f1,f2) f1:isSelf(f1)
       f1:isProxy(f2) access(f2,f2) access(f2,bob)
       access(f2,carol) f2:isSelf(f2) f2:isProxy(carol)
    ?  f2:isProxy(bob)
       access(alice,alice) access(alice,f1)
       access(bob,bob)
       access(carol,carol)
```

Figure 6.6: Example `config` part

The `config` part can contain:

- system knowledge facts to indicate the initial permissions. These facts must be derived from permission predicates, declared in the `system` part.
- public subject knowledge facts to indicate residue knowledge from earlier collaboration. These facts must be derived from subject knowledge predicates, declared in the `system` part.
- private subject knowledge facts to indicate initialization knowledge available to the subject. These facts must be derived from the private knowledge facts that

are declared in the *behavior* declaration that corresponds to the subject in their first argument.

The facts preceded by "?" indicate optional facts that will be maximized together with the behavior predicates of the subjects whose declaration is marked with ? in the subject part.

If a "?" is present in this part but not in the subject part, the SCOLL program defines a Knowledge Maximization Problem (Definition 19, Section 5.7.2). If a "?" is present in this part and in the subject part as well, the SCOLL program defines a configuration maximization problem (Definition 20, Section 5.7.2).

In the example of figure 6.6, the access() permission facts describe an initial access graph. The example further shows that f1 is initialized to forward to f2, who is initialized to forward to carol and optionally also to bob. Both forwarders are also initialized with explicit isSelf() knowledge.

It is considered bad practice to initialize a subject X with private knowledge about a subject Y, while X has no initial permission to Y. Doing so can either indicate that X has a powerful ability to identifty Y and uses that ability consistently to check every subject, or that a modeling error was made. Because the latter is more probable, the SCOLL syntax checker will generate a warning, if it detects a private knowledge relation in the *config* part that is not accompanied by an appropriate permission fact.

An exception to this rule will be made in section 6.7.2, for initial knowledge that models the parent-child relation.

6.2.6 Goal

The goal part consist of the reserved word goal, followed by a sequence of liveness possibilities and safety properties. Liveness possibilities are presented as knowledge facts. Safety properties are presented as knowledge facts preceded by an exclamation mark "!".

```
goal
        access(bob,carol)
    !   access(bob,alice)
```

Figure 6.7: Example goal part

All facts in this part must be derived from knowledge predicates, declared in earlier parts. All facts are represented with explicit (visible) first argument. The arguments in the facts must be declared in the *subject* part.

The example in figure 6.7 contains a liveness possibility to indicate that the pattern should not prevent bob from having or acquiring access to carol. The safety property requires the pattern to prevent bob from having or acquiring access to alice.

6.3 Kernel SCOLL Syntax

Figure 6.3 shows the syntax of the kernel language in EBNF form.

⟨Program⟩	::=	⟨Declare⟩ ⟨System⟩ ⟨Behaviors⟩ ⟨Subjects⟩ ⟨Config⟩ ⟨Goals⟩
⟨Declare⟩	::=	⟨DPermission⟩ ⟨DBehavior⟩ ⟨DKnowledge⟩
⟨System⟩	::=	`system` ⟨Rule⟩$^{+}$
⟨Behaviors⟩	::=	`behavior` ⟨Behavior⟩$^{+}$
⟨Subjects⟩	::=	`subject` ⟨Subject⟩$^{+}$
⟨Config⟩	::=	`config` ⟨ConfigFact⟩*
⟨Goals⟩	::=	`goal` (⟨Safety⟩ | ⟨Liveness⟩)*
⟨DPermission⟩	::=	`permission` : ⟨DeclarePred⟩*
⟨DBehavior⟩	::=	`behavior` : ⟨DeclarePred⟩*
⟨DKnowledge⟩	::=	`knowledge` : ⟨DeclarePred⟩*
⟨DeclarePred⟩	::=	⟨PredLbl⟩ "`/`" ⟨Arity⟩
⟨Behavior⟩	::=	⟨BehaviorID⟩ "{" ⟨Rule⟩* "}"
⟨Rule⟩	::=	⟨Body⟩ "=>" ⟨Head⟩
⟨Body⟩	::=	⟨Pred⟩*
⟨Head⟩	::=	⟨Pred⟩
⟨Pred⟩	::=	(⟨VarID⟩ "`:`")$^{?}$ ⟨PredLbl⟩ "`(`" (⟨VarID⟩ ("`,`" ⟨VarID⟩)*)$^{?}$ "`)`"
⟨Subject⟩	::=	("`?`")$^{?}$ ⟨SubjID⟩ "`:`" ⟨BehaviorID⟩
⟨ConfigFact⟩	::=	("`?`")$^{?}$ ⟨Fact⟩
⟨Fact⟩	::=	(⟨SubjID⟩ "`:`")$^{?}$ ⟨PredLbl⟩ "`(`" (⟨SubjID⟩ ("`,`" ⟨SubjID⟩)*)$^{?}$ "`)`"
⟨Safety⟩	::=	"`!`" ⟨Fact⟩
⟨Liveness⟩	::=	⟨Fact⟩
⟨Arity⟩	::=	["`1`" – "`9`"] ["`0`" – "`9`"]*
⟨BehaviourID⟩	::=	["`A`" – "`Z`"]$^{+}$
⟨VarID⟩	::=	["`A`" – "`Z`"] ["`0`" – "`9`""`a`" – "`z`""`A`" – "`Z`"]*
⟨PredLbl⟩	::=	["`a`" – "`z`"] ["`0`" – "`9`""`a`" – "`z`""`A`" – "`Z`""`.`"]*
⟨SubjID⟩	::=	["`a`" – "`z`"] ["`0`" – "`9`""`a`" – "`z`""`A`" – "`Z`"]*

The reserved words : `declare`, `permission`, `behavior`, `knowledge`, `system`, `subject`, `config` and `goal` cannot be used as ⟨PredLbl⟩ or ⟨VarId⟩.

Figure 6.8: Kernel SCOLL Syntax

6.4 KBM Semantics

We define a SCOLL program's denotational semantics in terms of Knowledge Behavior Models (KBMs, Definition 9 Section 5.7.1). First we define formally how some parts of a SCOLL program add to the definition of the KBM that represents its logical semantics. Then we explain how the whole program describes the safety problems defined in section 5.7.2.

Definition 21. *KBM Semsantics of a Kernel SCOLL program*
Let P_{SCOLL} be a SCOLL program, defined by its syntax as described in section 6.3, with the structural restrictions described in section 6.2.
The KBM $G = \langle S, P, V, R\rangle$ is the KBM semantics of P_{SCOLL} where:

S is the set of subjects declared in the `subject` *part of P_{SCOLL}.*

P is a set of predicate variables corresponding to the predicates declared in the `declare` *part of P_{SCOLL} augmented with the disjunct union of all behavior-specific predicates declared in the* `behavior` *part of P_{SCOLL}.*

V is the infinite set of variables identified by the subject variable identifiers that are allowed in SCOLL.

R is the set of implication formulas that correspond to:
***1.** The rules in the* `system` *part of P_{SCOLL} :*
$p_1(X_{1,1}, \ldots, X_{1,ar(p_1)}) \wedge \ldots \wedge p_n(X_{n,1}, \ldots, X_{n,ar(p_n)})$
$\quad\rightarrow p_{n+1}(X_{n+1,1}, \ldots, X_{n+1,ar(p_{n+1})})$
for every rule declared in the `system` *part :*
$\mathtt{p}_1\mathtt{(}\mathtt{X}_{1,1}\mathtt{,}\ldots\mathtt{,}\mathtt{X}_{1,ar(p_1)}\mathtt{)}\ \ldots\ \mathtt{p}_n\mathtt{(}\mathtt{X}_{n,1}\mathtt{,}\ldots\mathtt{,}\mathtt{X}_{n,ar(p_n)}\mathtt{)}$
$\mathtt{=>}\ \mathtt{p}_{n+1}\mathtt{(}\mathtt{X}_{n+1,1}\mathtt{,}\ldots\mathtt{,}\mathtt{X}_{n+1,ar(p_{n+1})}\mathtt{)}$
***2.** augmented with the disjunct union per subject s in S of the rules corresponding to the behavior rules of s :*
$p_1(s, X_{1,2}, \ldots, X_{1,ar(p_1)}) \wedge \ldots \wedge p_n(s, X_{n,2}, \ldots, X_{n,ar(p_n)})$
$\quad\rightarrow p_{n+1}(s, X_{n+1,2}, \ldots, X_{n+1,ar(p_{n+1})})$
for every rule declared in the `behavior` *part corresponding to the behavior of s :*
$\mathtt{p}_1\mathtt{(}\mathtt{X}_{1,2}\mathtt{,}\ldots\mathtt{,}\mathtt{X}_{1,ar(p_1)}\mathtt{)}\ \ldots\ \mathtt{p}_n\mathtt{(}\mathtt{X}_{n,2}\mathtt{,}\ldots\mathtt{,}\mathtt{X}_{n,ar(p_n)}\mathtt{)}$
$\mathtt{=>}\ \mathtt{p}_{n+1}\mathtt{(}\mathtt{X}_{n+1,2}\mathtt{,}\ldots\mathtt{,}\mathtt{X}_{n+1,ar(p_{n+1})}\mathtt{)}$

It is trivial to check that G is indeed a KBM, by comparing the definitions and the structural properties.

To define the KBM semantics of a kernel SCOLL program, we did not make use of the `config` part or the `goal` part of the program and we did not consider any maximization question marks in the program. These parts will now be used to define problems on the KBM (and their solutions).

Definition 22. ***Safety Problems in SCOLL***

Let P_{SCOLL} be a SCOLL program in which :

- no subject is marked with `?` in the `subject` part
- no fact is marked with `?` in the `config` part
- no liveness requirements are present in the `goal` part

Let G be the semantic KBM of the program.
Let C be the configuration of G consisting of the knowledge facts described in the `config` part of P_{SCOLL}.
Let X be the set of required safety properties described in the goal part of P_{SCOLL}
Then P_{SCOLL} defines the safety problem that is to decide if $X \subseteq Safe_G(C)$.

Definition 23. ***Practical Safety Problems in SCOLL***

Let P_{SCOLL} be a SCOLL program in which :

- no subject is marked with `?` in the `subject` part
- no fact is marked with `?` in the `config` part

Let G be the semantic KBM of the program.
Let C be the configuration of G consisting of the knowledge facts described in the `config` part of P_{SCOLL}.
Let X be the set of required safety properties described in the goal part of P_{SCOLL}
Let Y be the set of required liveness possibilities described in the goal part of P_{SCOLL}
Then P_{SCOLL} defines the practical safety problem that is to decide if $X \subseteq Safe_G(C)$ and $Y \subseteq Live_G(C)$.

Definition 24. ***Behavior Maximization Problems in SCOLL***

Let P_{SCOLL} be a SCOLL program in which :

- no fact is marked with `?` in the `config` part

Let G be the semantic KBM of the program.
Let C be the configuration of G consisting of the knowledge facts described in the `config` part of P_{SCOLL}.
Let X be the set of required safety properties described in the goal part of P_{SCOLL}
Let Y be the set of required liveness possibilities described in the goal part of P_{SCOLL}
Then P_{SCOLL} defines the behavior maximization problem that is to find the maximal sets of additional behavior facts for the `?`-marked subjects that ensure that $X \subseteq Safe_G(C)$ and $Y \subseteq Live_G(C)$.

Definition 25. ***Knowledge Maximization Problems in SCOLL***

Let P_{SCOLL} be a SCOLL program in which :

- no subject is marked with `?` in the `subject` part

Let G be the semantic KBM of the program.
Let C be the configuration of G consisting of the knowledge facts described in the `config` part of P_{SCOLL}.
Let X be the set of required safety properties described in the goal part of P_{SCOLL}
Let Y be the set of required liveness possibilities described in the goal part of P_{SCOLL}
Then P_{SCOLL} defines the knowledge maximization problem that is to find the maximal sets of `?`-marked knowledge facts in the `config` part P_{SCOLL} that still ensure that $X \subseteq Safe_G(C)$ and $Y \subseteq Live_G(C)$.

Definition 26. ***Configuration Maximization Problems in SCOLL***

Let P_{SCOLL} be a SCOLL program.
Let G be the semantic KBM of the program.
Let C be the configuration of G consisting of the knowledge facts described in the `config` part of P_{SCOLL}.
Let X be the set of required safety properties described in the goal part of P_{SCOLL}
Let Y be the set of required liveness possibilities described in the goal part of P_{SCOLL}
Then P_{SCOLL} defines the configuration maximization problem that is to find the maximal sets of additional behavior facts for the `?`-marked subjects and of `?`-marked knowledge facts in the `config` part that still ensure that $X \subseteq Safe_G(C)$ and $Y \subseteq Live_G(C)$.

6.5 The Complete SCOLL Language

A SCOLL program has the same six parts as a program in kernel SCOLL. The general purpose of each part is largely unchanged. Some restrictions in the kernel language will be removed or relaxed. This section describes these relaxations and their purpose. The translation of the newly allowed constructs to the kernel language will unambiguously define their semantics.

SCOLL is a language in evolution. Future extensions can use the same techniques of linguistic abstraction and syntactic sugar to extend and/or adapt the full language and will not require any changes in the implementation of the kernel language, unless for non-functional concerns (e.g performance, scalability). Apart from this practical advantage, the kernel language approach provides a simple way to discuss changes proposed to the language, in terms of the simpler kernel language.

In this section we describe the current version of SCOLL, which corresponds to the current version of the constraint based implementation of the language, called "SCOLLAR" that will be presented in chapter 7.

6.5.1 Multi-Headed Rules

We allow multi-headed system and behavior rules of the form:

$$
\begin{array}{l}
\texttt{p}_1\texttt{(V}_1\texttt{,...,V}_{k_1}\texttt{)}\ \texttt{...}\ \texttt{p}_n\texttt{(V}_1\texttt{,...,V}_{k_n}\texttt{)}\\
\quad \texttt{=>}\ \texttt{q}_1\texttt{(V}_1\texttt{,...,V}_{l_1}\texttt{))}\ \texttt{...}\ \texttt{q}_m\texttt{(V}_1\texttt{,...,V}_{l_m}\texttt{);}
\end{array}
$$

Both left hand side and right hand side of the rules are now conjunctions. The semicolon "`;`" is used as a delimiter between rules. "We translate such a rule into the kernel language as m different Horn clauses :

$$
\begin{array}{lll}
\texttt{p}_1\texttt{(V}_1\texttt{,...,V}_{k_1}\texttt{)}\ \texttt{...}\ \texttt{p}_n\texttt{(V}_1\texttt{,...,V}_{k_n}\texttt{)} & \texttt{=>} & \texttt{q}_1\texttt{(V}_1\texttt{,...,V}_{l_1}\texttt{)}\\
\texttt{p}_1\texttt{(V}_1\texttt{,...,V}_{k_1}\texttt{)}\ \texttt{...}\ \texttt{p}_n\texttt{(V}_1\texttt{,...,V}_{k_n}\texttt{)} & \texttt{=>} & \texttt{...}\\
\texttt{...} & \texttt{=>} & \texttt{...}\\
\texttt{p}_1\texttt{(V}_1\texttt{,...,V}_{k_1}\texttt{)}\ \texttt{...}\ \texttt{p}_n\texttt{(V}_1\texttt{,...,V}_{k_n}\texttt{)} & \texttt{=>} & \texttt{q}_m\texttt{(V}_1\texttt{,...,V}_{l_m}\texttt{)}
\end{array}
$$

For system rules, all predicates on the right hand site must be declared as permission or knowledge predicates, for the translation to be valid in the kernel language. For behavior rules, all predicates on the right hand side must be declared as `behavior` predicates, for the translation to be valid in the kernel language.

From now on we will use multi-headed rules when appropriate, as their translation to Horn clauses is straight forward and all other transformation rules will be independent.

6.5.2 Using Wildcards

All subject variables have rule scope. To indicate that two arguments should always be substituted by the same value, we use the same variable identifier for both arguments. To indicate that an argument should be substituted independently from all other arguments, we must make sure that the variable is unique in the rule.

We can make the latter more explicit and immediately visible by using the underbar character "_" as a wildcard, instead of a normal variable identifier. Upon preprocessing, every occurrence of "_" will be replaced by a different variable identifier that is unique in the rule.

For example this behavior rule:

```
did.sendTo(A,_) did.sendTo(B,_)
 => may.sendTo(A,B) may.return(_)
```

will be transformed into :

```
did.sendTo(A,V1) did.sendTo(B,V2)
 => may.sendTo(A,B) may.return(V3)
```

before being translated to the kernel language.

6.5.3 Explicit Refinement Rules

This section describes the extension of the `system` part with refinement rules: rules that derive general subject knowledge from specific subject knowledge and rules that derive specific behavior from general behavior. Their translation into the kernel language will be explained using a preprocessing step that removes the refinement rules.

We introduced the concept of behavior refinement in section 5.5 by means of an example of behavior depending on the invocation context.

The behavior predicates `may.returnFor` and `may.returnFor0` were used to refine the `may.return` behavior. The first one expresses the responder's intent to return a value for the invocations in which the invoker emits another value and to express requirements about the latter. The second one expresses the responder's intent to return a value for the invocations in which the invoker emits nothing.

The knowledge predicates `did.returnFor` and `did.returnFor0` refined `did.return` knowledge. The complementary knowledge `did.getFrom` was refined to `did.getFromFor` and `did.gettFromFor0`.

The refined behavior and knowledge predicates allowed us to express not only return behavior that depends on what the invoker sends in the same invocation (Section 5.5.4), but also the behavior of a proxy subject (Section 5.5.5) that respects such decisions made by its target.

The refinement approach was generalized in section 5.6.2, from which we repeat here only the conclusions :

1. General behavior implies refined behavior
2. Refined knowledge implies general knowledge
3. System rules should generate refined knowledge (more knowledge) from refined behavior (less behavior).

4. To refine a relied-upon subject's behavior: express the body of its behavior rules using more refined knowledge (stronger conditions) and/or express the head of its behavior rules using more refined behavior (applicable in less situations).

The first two conclusions above express refinement relations between the predicates and are independent of the actual system rules and behavior rules. We will allow the user to express these refinement relations explicitly as rules in the `system` part, by introducing rules of two new types:

1. Rules to generate more refined knowledge from less refined knowledge.
2. Rules to deduce more refined behavior from less refined behavior.

Doing so will allow the user to keep the unrefined system rules as they are and to add only new system rules for the refined predicates. The existing behavior declarations need no adaptations, except of course for the relied-upon subjects whose behavior is being refined.

```
declare
    ...
system
    B:may.return(Y)
     => B:may.returnFor0(Y) B:may.returnFor(X,Y);
    B:did.returnFor0(Y) => B:did.return(Y);
    B:did.returnFor(X,Y) => B:did.return(Y);
    ...
    ...B:may.return(Y) ... => ...
    ...
    ... => ... B:did.returnFor0(Y) ...
    ...
    ... => ... B:did.returnFor(X,Y) ...
    ...
behavior
    ...
    BEHAVIORi{
        ...
        ... => may.return(Y) ...
        ...}
    ...
subject
    s1 :BEHAVIORi1 ... sk :BEHAVIORik
config
    ... si:did.returnFor0(sj) ...
goal
    ...
```

Figure 6.9: Extract from a SCOLL program with refinement rules

An example will show how the refinement rules are preprocessed before being translated into kernel SCOLL. Consider the program extract in figure 6.9. The first rule in the `system` part is a behavior refinement rule, the next two rules are knowledge refinement rules.

The preprocessing will remove these rules from the `system` part and will make appropriate changes to the system rules that have an unrefined knowledge predicate in their body and/or a refined knowledge predicate in their head. In the `behavior` part, the rules that have the unrefined behavior predicate in their head will be changed. The `config` part will also be affected.

After preprocessing, the SCOLL program in Figure 6.10 does no longer contain the refinement rules. Every system rule that contains an unrefined behavior predicate is copied once for every refinement of the predicate. In that copy the unrefined predicate is replaced by the refined predicate. Since only the system rules generate this kind of knowledge, this guarantees that every refined behavior fact will generate at least the same knowledge as its unrefined version.

```
system
    ...
    ...rEmit(B Y) ... => ...
    ...rEmit0(B Y) ... => ...
    ...did.returnFor(B X(1) Y) ... => ...
    ...
    ... => ... rEmitted0(B Y) rEmitted(B Y) ...
    ...
    ... => ... did.returnFor(B X Y) rEmitted(B Y) ...
    ...
behavior
    ...
    BEHAVIORi{
        ...
        ... => rEmit(Y) rEmit0(Y) may.returnFor(X(1) Y) ...
        ...}
    ...
subject
    s1 :BEHAVIORi1 ... sk :BEHAVIORik
config
    ... rEmitted0(si sj) ...
    rEmitted(si sj)
goal
    ...
```

(1) The added variable must differ from all existing variables in the rule.

Figure 6.10: The equivalent extract without refinement rules

Every system rule that contains a refined knowledge predicate in its head will be extended, by adding the unrefined version of the predicate to the head of the rule. Since only the system rules generate this kind of knowledge, this guarantees that whenever a refined knowledge fact is generated, the unrefined version will be generated too. This means that every refined knowledge that is generated will generate at least the same behavior as its unrefined version.

Every subject rule that contains an unrefined behavior predicate in its head will be extended by adding every refined version of the predicate to the head of the rule. Since only behavior rules generate behavior, this guarantees that whenever an unrefined behavior fact is generated, every refined version of the fact is also generated.

Refined subject knowledge facts in the `config` part represent refined knowledge, left over from an earlier evolution (Section 5.4.6). These facts are not necessarily regenerated by the system rules. They must therefore be complemented by the unrefined version of the knowledge. Otherwise, refined left-over knowledge could have less effects on the generation of behavior (via the behavior rules) than its unrefined version would have.

All these preprocessing transformations ensure that unrefined behavior remains a safe over-approximation of refined behavior.

The bodies of the behavior rules need no preprocessing. Whenever a general knowledge fact triggers a behavior rule, its refined version will trigger the same rule indirectly, because the changes in the `system` part and the `config` part make sure that unrefined knowledge facts are always complemented by their refined versions.

Multiple refinements are preprocessed one by one. The order in which this preprocessing happens will only affect the order of the rules and of the predicates in the rules in the preprocessed program.

From the definition of the KBM semantics of a kernel program (Definition 21), we can directly conclude that the order of the rules and the order of the predicates in the body of the rules have no effect on the semantics of the program.

6.5.4 Safe Defaults

Unrestricted behavior safely approximates all behavior and is therefore the default behavior in SCOLL. In the `subject` part, subjects can be declared without a behavior. If so, it will have the default behavior.

```
declare
    permission :  ...
    behavior :  b1/i1 ... bn/in
    knowledge :  ...
system
    ...
behavior
    ...
    ...
    ...
subject
    ...
    alice
...
```

Figure 6.11: Default behavior in SCOLL

The translation is shown by example in figures 6.11 and 6.12. The default behavior can be declared in a single behavior rule that has an empty body and has every declared behavior predicate in its head, using the underbar "_" for every argument.

```
declare
   permission :  ...
   behavior :  b1/i1 ... bn/in
   knowledge :  ...
system
   ...
behavior
   ...
   DEFAULT {
      => b1(_,...,_) ... bn(_,...,_)}
subject
   ...
   alice :  DEFAULT
...
```

Figure 6.12: The equivalent kernel SCOLL

6.6 SCOLL Syntax

For completeness, figure 6.6 shows the syntax of SCOLL in EBNF form. The syntax only differs from the kernel language syntax in the following ways:

- Rule heads can now contain multiple predicates (at least one) and end with a semicolon.
- The underbar ("_") wildcard is introduced.
- The restriction that at least one behavior must be declared in the `behavior` part is removed, since subjects can use the default behavior.
- Subjects specify their behavior optionally in the `subject` part.

6.7 Possible Extensions

Many other extensions to SCOLL are possible and useful. Some of them will be presented in this section, most of them are in a very early design stage. They will make it into SCOLL if and when the need arises from experience by the SCOLL users community, depending on the time and resources available in the open source SCOLL development group and on the priorities put forward by users and developers alike.

6.7.1 Expressing Refinement Partial Orders

Section 5.6.2 explained how the introduction of a *nil* element, a $\perp$ element (bottom) and a partial order can be used to refine the `may.return` predicate and how this can be generalized.

⟨Program⟩	::=	⟨Declare⟩ ⟨System⟩ ⟨Behaviors⟩ ⟨Subjects⟩ ⟨Config⟩ ⟨Goals⟩
⟨Declare⟩	::=	⟨DPermission⟩ ⟨DBehavior⟩ ⟨DKnowledge⟩
⟨System⟩	::=	`system` ⟨Rule⟩$^+$
⟨Behaviors⟩	::=	`behavior` ⟨Behavior⟩*
⟨Subjects⟩	::=	`subject` ⟨Subject⟩$^+$
⟨Config⟩	::=	`config` ⟨ConfigFact⟩*
⟨Goals⟩	::=	`goal` (⟨Safety⟩ | ⟨Liveness⟩)*
⟨DPermission⟩	::=	`permission` : ⟨DeclarePred⟩*
⟨DBehavior⟩	::=	`behavior` : ⟨DeclarePred⟩*
⟨DKnowledge⟩	::=	`knowledge` : ⟨DeclarePred⟩*
⟨DeclarePred⟩	::=	⟨PredLbl⟩ "/" ⟨Arity⟩
⟨Behavior⟩	::=	⟨BehaviorID⟩ "{" ⟨Rule⟩* "}"
⟨Rule⟩	::=	⟨Body⟩ "=>" ⟨Head⟩ ";"
⟨Body⟩	::=	⟨Pred⟩*
⟨Head⟩	::=	⟨Pred⟩$^+$
⟨Pred⟩	::=	(⟨VarID⟩ ":")$^?$ ⟨PredLbl⟩ "(" (⟨VarID⟩ ("," ⟨VarID⟩)*)$^?$ ")"
⟨Subject⟩	::=	("?")$^?$ ⟨SubjID⟩ (":" ⟨BehaviorID⟩)$^?$
⟨ConfigFact⟩	::=	("?")$^?$ ⟨Fact⟩
⟨Fact⟩	::=	(⟨SubjID⟩ ":")$^?$ ⟨PredLbl⟩ "(" (⟨SubjID⟩ ("," ⟨SubjID⟩)*)$^?$ ")"
⟨Safety⟩	::=	"!" ⟨Fact⟩
⟨Liveness⟩	::=	⟨Fact⟩
⟨Arity⟩	::=	["1" – "9"] ["0" – "9"]*
⟨BehaviourID⟩	::=	["A" – "Z"]$^+$
⟨VarID⟩	::=	["A" – "Z"] ["0" – "9""a" – "z""A" – "Z"]*
⟨PredLbl⟩	::=	["a" – "z"] ["0" – "9""a" – "z""A" – "Z"]*
⟨SubjID⟩	::=	["a" – "z"] ["0" – "9""a" – "z""A" – "Z""."]*

The reserved words : `declare`, `permission`, `behavior`, `knowledge`, `system`, `subject`, `config` and `goal` cannot be used as ⟨PredLbl⟩ or ⟨SubjID⟩.

Figure 6.13: SCOLL Syntax

Expressing predicate refinement this way can result in a very concise notation of the system rules. A single rule suffices to model collaborative propagation in capability systems:

```
A:may.call(B,+X) B:may.returnCall(+X,+Y)
access(A,B) access(A,+X) access(B,+Y)
=> A:did.call(B,+X,+Y) B:did.returnCall(+X,+Y)
   access(B,+X) access(A,+Y)
```

The variable identifiers `X` and `Y` are prefixed with a + sign to indicate that they range over the extended set that contains not only the subjects but also the element nil that means "no subject" and the element $\perp$ that means: "unspecified" (either a subject or nil).

When translating to the kernel language, all "+" signs are removed (the scope of the variables must be reduced to the set of subjects). The rules must then be duplicated separately for the nil and $\perp$ elements and all refinement rules implied by the partial order must be made explicit.

Before introducing refinement by partial order into the language, it will must be decided what partial order(s) will be supported and how the partial order will be expressed. To support a single fixed partial order like the flat semi-lattice in figure 5.6, the introduction of the `nil` keyword and a reserved symbol to indicate the $\perp$ element would suffice.

6.7.2 Support for Parenthood and Endowment

The parent-child relation between the subjects can be expressed in the `config` part, using a permission predicate declared in the `declare` part. The system rules can then express propagation by endowment and behavior (section 5.4.3) using these permissions.

To make sure that the behavior is only taken into account for children that are actually created, another permission must be declared (e.g. `alive/1`) and every system rule must include this permission for every variable in the rule. All these extra predicates and rules will clutter up the SCOLL program. They can distract the user in his task to safely approximate entity behavior into subjects.

We propose to predefine the permission predicates `child/2` and `alive/1`, the behavior predicates `may.create/2` and `may.endow/3`, and the knowledge predicates `did.create/2` `did.endow/3` and `was.endowed/2`, and the following implicit and predefined system rules to express propagation by parenthood and endowment:

```
alive(P) P:may.create(C) child(P,C)
 => access(P,C) P:did.create(C)

P:did.create(C) => alive(C)

P:did.create(C) access(P,X) alive(X) P:may.endow(C,X)
=> access(C,X) P:did.endow(C,X) C:wasEndowed(X)
```

The `child()` predicates indicate which subject can create which other subject. This relation is static and must be expressed in the `config` part of the program.

Every subject that models at least one entity that is alive in the initial configuration must have the `alive` permission in the `config` part.

Preprocessing a SCOLL program that contains at least one occurrence of an `alive` permission in the configuration or of a `may.create` or `may.endow` fact in the `behavior` part, will require the following transformations:

1. For every variable `V` : add the `alive(V)` predicate in the body of every rule that contains the variable `V`.
2. Add the system rules for parenthood and endowment, given above.
3. Warn the user about the subjects in the `subject` part that are not `alive` in the `config` part.

6.7.3 Support for Creation and Aggregation

Even with the support of predefined predicates and rules for parenthood and endowment, the user mapping code to SCOLL must at the same time extract behavior, aggregation and the parenthood relation from the code.

It would be better if the user could model the code into behavior in one time (the creation behavior included), and model the aggregation in another time. This would not only make it easier to avoid modeling mistakes, but also allow the user to experiment with the level of detail in the aggregation relation, independent of behavior.

We can separate both aspects if we allow the parent subject to specify the behavior of the child that is created. Unspecified behavior would be mapped automatically to unrestricted default behavior.

An extra part in structure of a SCOLL program could be dedicated to express the aggregation relation. In that part, the children that result from successful `may.create` behavior of one subject would be associated with a new or an existing subject. By default, as a safe but crude approximation, every child that gets created would be aggregated into a single subject, regardless of its parent.

6.7.4 Goal Refinements

Goals are expressed as knowledge facts that either must be attainable or should not be attainable in the KBM that corresponds to the SCOLL program. They express very simple constraints on the set of knowledge facts that can be reached in the KBM. The safety properties express an upper bound to propagation (what should not be reached), while the liveness possibilities express a lower bound to propagation (what should not be prevented).

More elaborate constraints could be used to express conditional safety properties and thus define new interesting safety problems on KBMs. The subject will be revisited in chapter 9, but a definite proposal and implementation is left as future work.

6.7.5 Syntactic Sugar for Predicate Declarations

We expect that the more or less intensive use of the language by a group of users may spawn suggestions for improving the usability of the `declare` part using syntactic sugar. However, to minimize the chance of modeling errors, we advise against making the declarations optional, even if they can be inferred from the other parts of the SCOLL program.

The explicitly declared type and arity of a predicate not only allow us to check if a system rule or a behavior rule is valid, it also allowed us to introduce new types of rules, without necessarily adding an explicit type system for the rules (Section 6.5.3).

6.7.6 Disjunctions in Rule Bodies

When two or more conditions lead to the same behavior, a behavior rule must be modeled for every one of these condition. An extension of SCOLL could allow disjunctive conditions, in behavior rules and/or in system rules. The transformation of a rule with a disjunctive body into multiple rules without disjunction would be straight forward, as is shown by this example:

$(c_{1,1}() \ \dots \ c_{1,n}())$ `or` $(c_{2,1}() \ \dots \ c_{2,m}())$ `=> p()`

would be decomposed into:

$c_{1,1}() \ \dots \ c_{1,n}()$ `=> p()`

$c_{2,1}() \ \dots \ c_{2,m}()$ `=> p()`

6.7.7 Expressing Behavior Restrictions with Negated Predicates

Behavior can be expressed directly, but behavior restrictions can only be expressed indirectly. In SCOLL, the negation of a predicate cannot be expressed directly: there is no negation operator. For instance, a conditional behavior restriction can only be expressed by making sure that all the behavior rules include the complementary condition in their body.

For example, to express:
```
   special(X) => ¬ may.grantTo(Y,X)
```
we have to change every rule in the behavior of the form:
```
   condition1(...) ... conditionN(...) => may.grantTo(Y,X)
```
into:
```
   normal(X) condition1(...) ... conditionN(...)
    => may.grantTo(Y,X)
```

An extension of SCOLL could in principle provide this facility as a linguistic abstraction, but that would violate the language design rule we derived earlier: that every condition in a behavior rule should be explicit. Forgetting a condition will only make the behavior rule fire more easily and still guarantee a safe approximation of the behavior.

6.7.8 Expressing Behavior Conditional on Negated Predicates

KBMs are monotonic : more knowledge never leads to less behavior and more behavior never leads to less knowledge. The reason is that actual knowledge and behavior facts in a KBM express the possibility of knowledge and behavior in an actual system that is safely approximated by the KBM.

The impossibility of a knowledge or behavior fact cannot be stated, it can only be derived as a safety property: if every possibility is taken into account, then what remains is impossible.

Positive knowledge facts indicate the possibility that the knowledge is attained. Negative knowledge facts can only indicate the possibility that contradicting knowledge is attained. We must be careful not to infer the impossibility of a fact from the possibility of a contradicting fact. Moreover, at least one of both will be possible: the fact or the contradiction of the fact.

Consider a behavior rule of the following form:

```
¬ knowledge(X) => behavior(X)
```

This rule cannot express `behavior()` on the condition of not having the relation `knowledge()` with `X`. It can only expresses `behavior()` on the condition of having a relation with `X` that contradicts `knowledge(X)`. Both `knowledge(X)` and `¬ knowledge(X)` represent possible knowledge. While in every possible execution of the program these facts exclude each other, their possibilities do not exclude each other.

Except maybe for static private knowledge (used but not generated by behavior rules), we advise against the introduction of negated knowledge predicates in the language, unless strong support can be provided to the user, that will help him/her avoid modeling mistakes.

6.8 Modeling in SCOLL

In this section we revisit the relation between the code in a programming language and its model in SCOLL. This relation is important in both directions. A given program can be mapped to a SCOLL pattern to analyze and verify its required safety properties and to suggest changes (additional restrictions) if the requirements are not satisfied. The results of the analysis in SCOLL (the solutions to the problem declared in the SCOLL program), are modified versions of the original SCOLL pattern, which must then be translated back to code in the programming language.

Figure 6.14 gives a schematic overview of the components in the translation process. It is not practical to translate to and from the program's actual operational semantics. Abstract interpretation of the source code to a simplified state machine that safely approximates the reachable states would be preferred. The authority in the model can then be defined in terms of predicates about the simplified state. Techniques for transforming to and from source code will be discussed as future work in chapter 11.

Translations to and from program code may not always be required. SCOLL is a modeling language that is useful as such, for instance to compare different protection systems or alternative behaviors, without actually making the translation to or from code. Even so, the translation in either direction will be important to developers of secure software.

This work does not include an automatic or semi-automatic translator to or from any specific programming language, nor does it describe the design principles for such a translator. That is left as future work. This section will only give an informal introduction to the most important issues developers need to consider when making the translation manually.

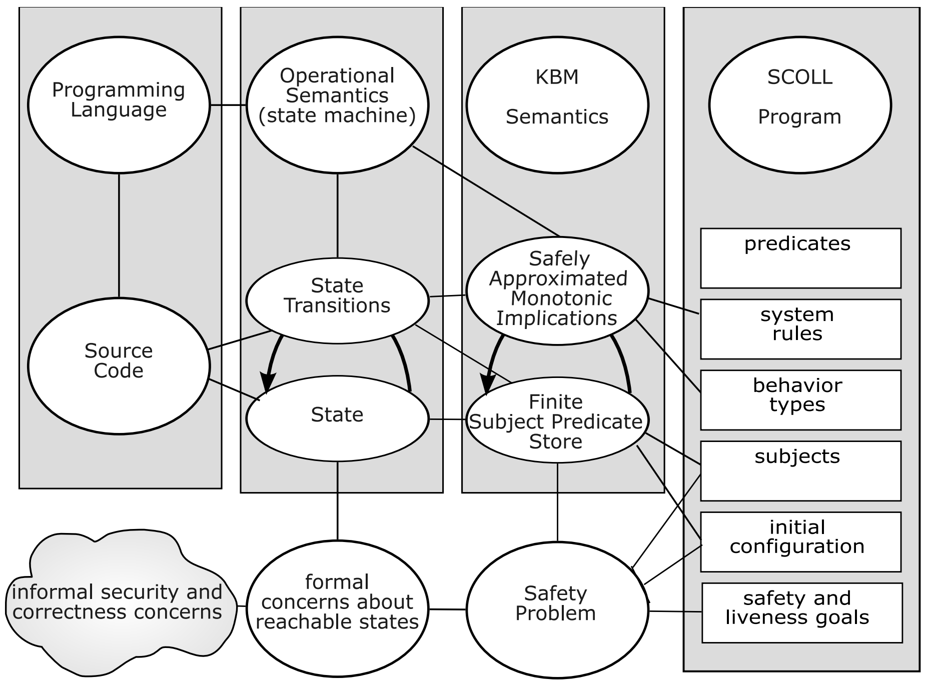

Programming Language
Operational Semantics (state machine)
KBM Semantics
SCOLL Program
predicates
State Transitions
Safely Approximated Monotonic Implications
system rules
Source Code
behavior types
State
Finite Subject Predicate Store
subjects
initial configuration
informal security and correctness concerns
formal concerns about reachable states
Safety Problem
safety and liveness goals

Figure 6.14: Translation between SCOLL and actual code

6.8.1 Modeling Authority Propagation

To analyze boundaries for authority propagation, we need to define first what the entities are, from the following considerations:

- Entities can have and use permissions and authority.
- Entities can interact with other entities.
- Entities can propagate permissions and authority by interacting with other entities.
- Entities can often create other entities.

For programs in a pure object oriented language, the object instances are the best candidates. For procedural languages, the runtime instances of the procedure's closure can be chosen.

System rules will be used to express the mechanisms that can cause propagation of permissions and authority. To construct a valid model, all such mechanisms must be considered and mapped to a system rule. The mandatory preconditions for a mechanism can be modeled as preconditions in a system rule. Forgetting or relaxing a precondition does not invalidate the model, but can render it less accurate. Forgetting or underestimating an effect of a mechanism (or a complete mechanism) will render the model invalid.

If a certain mechanism has mandatory prerequisites the user wants to model, he/she must find out for each of the prerequisites whether they are permissions or behavior preconditions.

It is easy to recognize the difference: behavior preconditions depend on the code that will be executed by a programmed entity, while permissions are independent of such code. If in doubt, consider only entities with completely unrestricted behavior and ask yourself the question: “Can this precondition be *not* met and will that prevent the propagation of authority ?”. If the answer is yes, it means that the precondition is independent of behavior and is therefore a permission.

Permissions may not be directly accessible to the entity. That is the case for instance, when a runtime entity can only know if it has read permission to a file by actually trying to read it. This test is not accurate though, because it tests read authority, instead of read permission. Since permissions do not necessarily guarantee authority, a failure of the test is not conclusive, unless it is known that the file’s behavior cannot or will not interfere with the permission. To discover what the actual permissions are that govern a mechanism for authority propagation, a feasible approach is to start with a single general *access* permission and refine it when necessary.

Let us apply the approach to a memory-safe language like Java: what are the mechanisms for authority propagation? Objects can send messages to other objects and pass references as arguments to the message and as return values from the message. Functions can be invoked with input arguments referring to other functions and objects and can return such references too. All the references are unforgeable because the language is memory safe. Access to a reference is a permission, because whatever the object’s behavior, it cannot invoke a function it has no reference to and it cannot send a message to an object it has no reference to.

Concerning the effects of authority propagation, we distinguish again between two types: permissions and entity knowledge. Both concern the state of an entity, but the knowledge part is directly accessible by the entity, while the permission part is not. Everything an entity can possibly learn from its direct involvement in a propagation mechanism, should be modeled as knowledge that becomes available to the entity to adapt its behavior. Section 6.8.9 will explain how to model behavior.

The following sections explore several aspects of authority propagation and show how they affect the `system` part of a SCOLL program.

6.8.2 Authority Propagation in the Presence of Global State

If a language provides variables with global scope, these variables provide an alternative mechanism for authority propagation that does not involve invocation or message sending. Every global variable can act as a channel that can be used by every entity to propagate access. We can model this mechanism as an extra system rule that does not require access between the emitter and the collector:

```
access(A,X) A:may.makeGlobal(X) B:may.useGlobal(B)
 => access(B,X);
```

The emitter still has the choice about what permission it will emit and the initiator of an interaction still has the choice about what subject to interact with (if it chooses not to use the global variables, but only invocation based collaboration). However, a confinement strategy that depends on interposition of relied-upon subjects with restricted behavior will fail, because the collaborating subjects do not need access to each other to propagate authority.

6.8.3 Authority Propagation in the Presence of Ambient Authority

Ambient authority is authority that is available to every entity. If a language provides ambient authority, we can model that as a unary permission, e.g.:

```
access(A,X) ambient(B) A:may.sendTo(B,X) B:may.receive()
 => access(B,X);
access(B,Y) ambient(B) A:may.getFrom(B) B:may.return(Y)
 => access(A,Y);
```

Global state is an extreme form of ambient authority, because the set of entities that can become globally accessible is not restricted. Global state is more permissive than ambient authority because it can be derived from it by adding the following rule:

```
globalVar(V) A:may.sendTo(V,X) => ambient(X)
```

From this example we learn that we can consider a partial order of *permissiveness* between systems: if one system can always be expressed in SCOLL with a subset of the system rules of another system, the latter system is more permissive.

6.8.4 Authority Propagation via Channels

The multi-paradigm programming language Oz has logic variables and stateful cells. Entities that interact with each other can not just propagate access to other entities by invocation, but can also share mutable cells or logic variables that establish a channel for propagation without invocation.

We can model this form of propagation by modeling channels as subjects and introducing a `may.channel/1` behavior predicate to indicate channel behavior, the `may.putOn/3` and `may.getFrom/2` behavior predicates for the behavior that uses the channel. In the example rules, the knowledge predicates are not considered.

```
access(A,C) access(A,X) A:may.putOn(C,X) C:may.channel()
=> access(C,X);
access(A,C) access(C,X) A:may.getFrom(C) C:may.channel()
=> access(A,X);
```

We can drop the extra rules and only use invocation based rules, if we model using a channel as just another form of invocation. The channel's behavior can be modeled as a passive subject that never initiates an interaction, but always emits and collects as a responder to an interaction.

6.8.5 Authority Propagation and the Principle of Attenuation

The principle of attenuation states that: "No subject should be able to delegate rights (permissions) it does not have". Since the holder of a permission is indicated as the first argument of a system knowledge predicate, we can express this principle as a restriction on its system rules in SCOLL.

Every system rule that has a permission in the head, should include that same permission (possibly held by another subject) in its body. The formal requirement is given in definition 27.

Definition 27. *Attenuation*
A kernel SCOLL system rule ***respects attenuation*** *if and only if it has the form:*
$\texttt{condition}_1 \ldots\ \texttt{condition}_n\ \texttt{=>}\ \texttt{permission}_p\texttt{(C}_1\texttt{,A}_2\texttt{,}\ldots\texttt{,A}_m\texttt{)}$
and the head is included in the body of the rule, modulo its first argument variable:
$\exists\, 1 \leq i \leq n : \texttt{condition}_i = \ \texttt{permission}_p\texttt{(D,A}_2\texttt{,}\ldots\texttt{,A}_m\texttt{)}$

A SCOLL system part ***respects attenuation*** *if and only if, after translation into kernel SCOLL, every rule with a permission in the head respects attenuation.*

6.8.6 Authority Propagation and the Granovetter Property

In this section we express the principle : ***only connectivity begets connectivity*** that was introduced in section 4.3.4. It is a stronger version of the attenuation principle, which adds the restriction that delegation is only possible between two entities if one of them has access to the other one.

The principle is sometimes referred to as the *Granovetter property*. Granovetter was a sociologist who studied the evolution in the topology of interpersonal relationships, as people introduce people they know to each other. In a generalized form, independent of the actual permission used in capability systems, this principle is formalized in definition 28.

Definition 28. *The Granovetter Property*
A kernel SCOLL system rule ***respects the Granovetter property*** *if and only if it has the form:*
$\texttt{condition}_1 \ldots \texttt{condition}_n\ \texttt{=>}\ \texttt{permission}_p\texttt{(A}_1\texttt{,A}_2\texttt{,}\ldots\texttt{,A}_m\texttt{)}$
$\exists\, 1 \leq i \leq n : \texttt{condition}_i = \ \texttt{permission}_p\texttt{(D,A}_2\texttt{,}\ldots\texttt{,A}_m\texttt{)}$
and $\exists\, 1 \leq k \leq n : (\texttt{condition}_k = \texttt{permission}_q\texttt{(A}_1\texttt{,}\ldots\texttt{,D,}\ldots\texttt{)}$
or
$\texttt{condition}_k = \texttt{permission}_q\texttt{(D,}\ldots\texttt{,A,}_1\texttt{,}\ldots\texttt{))}$

A SCOLL system part ***respects the Granovetter property*** *if and only if, after translation into kernel SCOLL, every rule with a permission in the head respects the Granovetter property.*

It is clear from this definition that a system with global variables (Section 6.8.2) does not respect the Granovetter property, while the system of section 6.8.4) that supported channels does.

6.8.7 Authority Propagation and Collaboration

To allow us to build a confinement strategy based on the restricted behavior of relied-upon subjects, the system has to consult the behavior of every entity that is involved in authority propagation.

Examples of such strategies will be given in chapter 8.

We can define collaboration formally as a property of SCOLL systems, if we interpret the “involvement” of a subject to mean: one of its permissions is required as a precondition for authority propagation.

Definition 29. *Collaborative Systems*
A kernel SCOLL system rule ***consults behavior*** *if and only if it has the form:*
$\texttt{pred}_1 \ldots \texttt{pred}_n \texttt{ => } \texttt{pred}_{n+1}$
such that $\forall 1 \leq i \leq n+1 : \texttt{pred}_i = \texttt{permission}_p\texttt{(A}_1\texttt{,A}_2\texttt{,}\ldots\texttt{,A}_m\texttt{)}$
$\Rightarrow \exists 1 \leq j \leq n : \texttt{pred}_j = \texttt{behavior}_j\texttt{(A}_1\texttt{,X}_2\texttt{,}\ldots\texttt{,X}_l\texttt{)}$

A kernel SCOLL system rule ***respects choice*** *if and only if it has has the form:*
$\texttt{pred}_1 \ldots \texttt{pred}_n \texttt{ => } \texttt{pred}_{n+1}$
such that $\forall 1 \leq i \leq n+1 : \texttt{pred}_i = \texttt{permission}_p\texttt{(A}_1\texttt{,A}_2\texttt{,}\ldots\texttt{,A}_m\texttt{)}$
$\Rightarrow \exists 1 \leq j \leq n : \texttt{pred}_j = \texttt{behavior}_j\texttt{(A}_1\texttt{,}\ldots\texttt{,X,}\ldots\texttt{)}$
and $\exists\, 2 \leq k \leq m : \texttt{X} = \texttt{A}_k$

A SCOLL system part ***is collaborative*** *if and only if, after translation into kernel SCOLL, every rule respects choice.*

6.8.8 Modeling Authority Propagation in Capability Systems

We have described the characteristics of object capability systems in section 4.3. The characteristics can be expressed as constraints on the system rules in SCOLL, which can in turn be used to check if the modeled system indeed implements a capability system.

Two kinds of constraints were mentioned in section 4.3.4. The first one is about making sure that enough permissions are required: only connectivity begets connectivity. The second one requires the behavior of the subjects to be consulted in a way that allows them to make the appropriate decisions. The emitter chooses what authority it will emit and the initiator chooses what subject it will interact with.

These constraints can now be formalized, combining collaborative systems with the Granovetter property:

Definition 30. *Object Capability Systems*
A SCOLL system part ***models object capabilities*** *if and only if:*

1. *it respects the Granovetter property*

2. *it is collaborative*

6.8.9 Modeling Behavior

SCOLL allows the user to express approximate behavior per rule. To model an entity's behavior, the developer only has to safely approximate every statement or method as a separate behavior rule.

To safely approximate an entity's behavior :

- Every statement in the entity's code that can propagate authority must be modeled by at least one behavior rule. Multiple statements can be mapped into a single behavior rule though.

- The head of the behavior rule must specify behavior that is an over-estimation of the behavior expressed in the statement.

- The body of the behavior rule must be a condition that is an under-estimation (weaker, easier to meet) of the precondition for the statement in the entity's code. It is always safe to drop the condition in the rule, because that will be interpreted as always *true*.
- Multiple statements can be mapped to the same rule if the preconditions for the statements separately imply the preconditions of the rule and if the postconditions of the rule imply the postconditions of the statements.
- Multiple entities can be modeled by the same subject, using the technique of aggregation. The behavior of this subject then contains (a safe approximation of) the union of all behavior rules that would be modeled separately. For instance, if one of the entities has access to itself and emits itself as a responder, the aggregated subject should do so too.
- When entities are aggregated into a subject, the `config` part (representing the initial configuration) should contain knowledge predicates that reflect the initial knowledge of and about all the entities, as knowledge of and about the aggregated subject. A very easy and practical us of aggregation is to model all offspring that can ever be created by an entity together with the entity itself as one subject.
- If in doubt, model the entity as an unrestricted or unknown subject, either by making a single behavior rule with an empty body that generates all possible behavior predicates, or simply by not stating the subject's behavior at all and using the safe default behavior instead.

6.9 Example : Inescapable Interposition

This section gives a first complete example of a SCOLL program, modeling a capability system. More will follow in chapter 8. The example is chosen because it shows how an appropriate application of aggregation can help to attain a safe, simple, and accurate model.

6.9.1 Overview

This example expresses a safety problem. Four entities and their offspring are involved. Two of them, `alice` and `bob`, have behavior we do not want to rely on. The other two, `proxyAlice` and `proxyBob`, have to keep the former two from getting direct access to each other. We will call `alice` and `bob` *unknown* because there is no behavior restriction we can (or want to) rely on that we will model in our SCOLL pattern.

We can assume that `alice` and `bob` are not connected to each other because our language allows us to load both entities without providing them any (ambient) authority. In the initial configuration, both unknown subjects have access to the other one's proxy: `bob` to `proxyAlice` and `alice` to `proxyBob`. The proxies have access to their target initial knowledge about their target: `proxyBob`'s target is `bob` and `proxyAlice`'s target is `alice`. All have access to themselves. The initial access graph is shown in figure 6.15.

The behavior of `proxyBob` should forward all messages to `bob`, but make sure that only data passes directly. If `alice` has a capability, e.g. access to a subject `a1`, and she wants to pass it to `bob`, she has to use `proxyBob` to forward it. The proxy to `bob` will not simply pass that capability on to `bob`, but do the following:

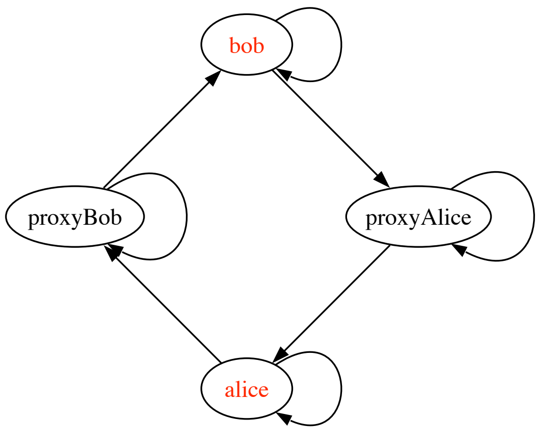


Figure 6.15: The initial configuration for the example

1. create a new entity `proxyA1` with the same proxy behavior.
2. endow it with access to `a1` and with knowledge that `a1` is its target
3. forward `proxyA1` to `bob` (instead of forwarding `a1` to him).

The same goes when `bob` returns a value for `proxyBob` to return to `alice`. Also, in the other direction, when `bob` wants to emit or collect a capability to/from `alice`, `proxyAlice` will always first wrap all its arguments.

The code for the relied-upon subjects `proxyAlice`, `proxyBob`, and their offspring is shown in figure 6.16.

Function `MakeProxy` takes a target entity (procedure) as input, and makes a forwarder to that entity. The procedure is assumed to take a single message argument: a record of the form `label(x:X ... z:Z)`. The message format is checked to be conform.

The proxy will actually forward another message of the same form, but with unbound logic variables: `label(x:_ ... z:_)`, that was constructed with the function `Record.clone`. Then, in a series of parallel threads (the Mozart implementation of Oz has very light threads), the procedure waits for the invoker or the responder to bind the variables (testing by `Record.waitOr` will cause the thread to block until at least one of the variables is bound).

If the argument is bound to an integer, the corresponding variable is bound to the same integer. If the argument is bound to a procedure, a new proxy is created that will target the procedure, and is bound to the corresponding variable in the other message.

Whether a position in the message is used for input or output of the message will be up to the unknown entities. A complete series of proxies can be chained like this, but the variables will always be bound by the unknown entities, or they will be left unbound, but then both entities will have different (not unified) logical variables in the message.

The `Target` variables for `ProxyAlice` and `ProxyBob` will be modeled as initial permissions `access(proxyAlice,alice)` and `access(proxyBob,bob)` and as initial private knowledge `proxyAlice:target(alice)` and `proxyBob:target(bob)`.

The untrusted entities `alice` and `bob` get initial permissions: `access(bob,proxyAlice)` and `access(alice,proxyBob).`

The language is relied upon to *not* provide ambient authority for `MakeUnknown` instances and thereby prevent them form sharing static variables.

The procedure propagating behavior of the proxies is to wrap every input procedure value in a forwarder before making it available to the target, and to wrap every output procedure value before making it available to the sender of the message.

Important: Notice that no unbound variables are shared ever between the sender and the target of the message. Doing so would create a direct communication channel between them, which has to be avoided.

The proxies create only new proxies, thereby endowing them with a target that is provided by the sender or by the receiver of the message. The procedure values that `Bob` binds to an input or output argument in a message to or from `ProxyAlice`, will be wrapped in a proxy that targets that value. The same goes for `Alice`.

6.9.2 Aggregating by Clan and by Target

Both unknown subjects have the possibility to create offspring at will and to introduce their offspring to each other, and to pass them over to the other clan, wrapped in a proxy instance. To approximate the whole program with a finite set of subjects, we need to aggregate most of them. Even after one invocation that involves wrapping the input and output arguments, the configuration already doubles in size, as is shown in figure 6.17.

Fortunately, the most appropriate way to aggregate the entities is not hard to find. It is indicated by the grayed areas in figure 6.17: first aggregate the unknown entities with their offspring (clan) and then aggregate the proxies according to their (aggregated) target.

Let us find out first what the aggregation implies in terms of behavior. Aggregating the behavior of unknown subjects with their offspring is never a problem, because they already have maximal unrestricted behavior and their offspring cannot add anything to that. The proxies have restricted behavior, but we can rely on them to create only offspring with behavior that is restricted in the same way.

We have to check what the aggregation means in terms of initial knowledge and permissions. The access configuration is simple, as we start the configuration with no offspring. The parent child relation can be directly derived from the aggregation: the unknown subjects are their own child, the proxy subjects are each other's child.

Figure 6.18 shows the SCOLL program. The aggregation relation is reflected in the `child()` facts of the initial configuration. The only goal of the program is: make sure that no instance of the `alice` clan ever gets direct access to an instance of the `bob` clan, or vice versa.

The SCOLL program in figure 6.18 describes a safety problem. The result of the safety problem will be shown in chapter 8.

6.10 Evaluation

Let us check if SCOLL lives up to the expectations we listed in section 6.1.

```
declare

fun{MakeProxy Target}
   proc{$ Msg}
      if {List.all {Label Msg}|{Arity Msg}
          fun{$ A} {IsAtom A}
          end}
      then
         NewMsg = {Record.clone Msg}
      in
         {Target NewMsg}
         {Record.zip Msg NewMsg
          fun{$ X Y}
             thread
                V1 V2 in
                case {Record.waitOr X#Y}
                of 1 then V1=X V2=Y
                [] 2 then V1=Y V2=X
                end
                if {IsInt V1} then V1 = V2
                elseif {IsProcedure V1}
                andthen {ProcedureArity V1}==1
                then V2  = {MakeProxy V1}
                end
             end
          end
          _}
      else raise messageNotConfom end
      end
   end
end

[MakeUnknown] = {Load ['x-oz://untrusted/unknown.ozf']}

Alice = {MakeUnknown [ProxyBob]}

Bob = {MakeUnknown [ProxyAlice]}

ProxyAlice  = {MakeProxy Alice}

ProxyBob  = {MakeProxy Bob}
```

Figure 6.16: Oz-E code for the proxy’s behavior

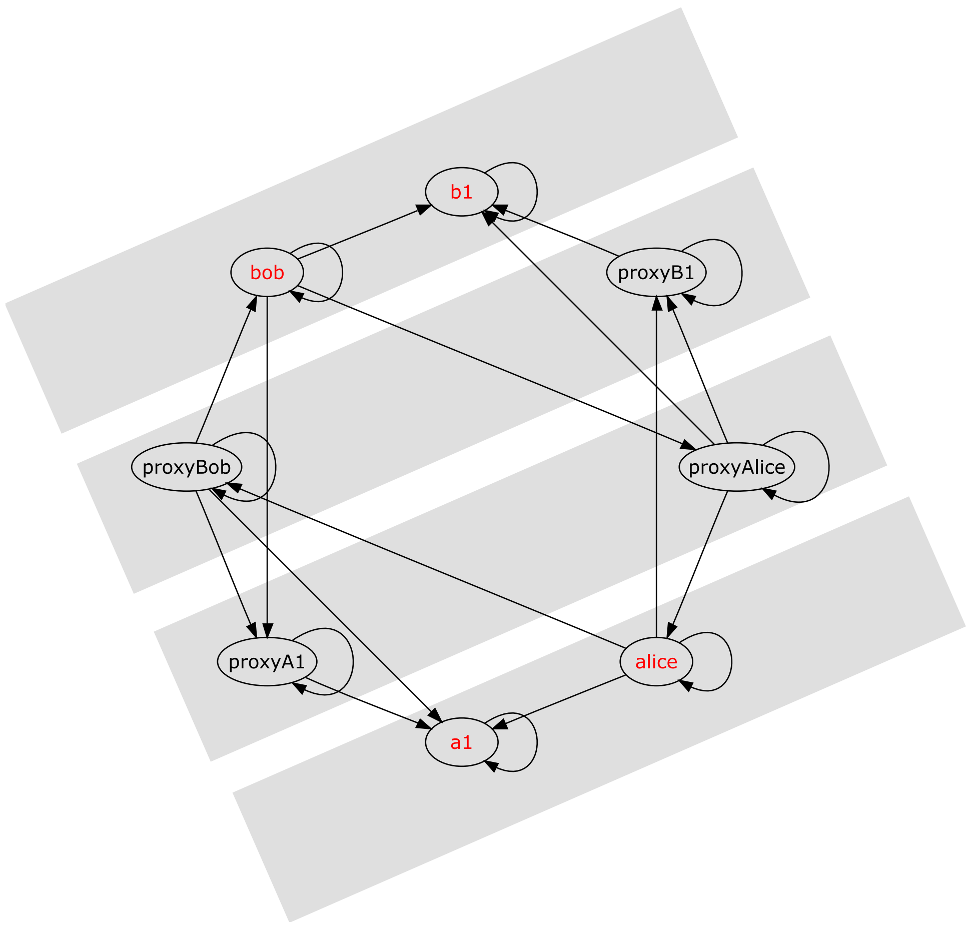


Figure 6.17: The configuration after a single invocation: grayed areas indicate which entities will be aggregated.

```
declare
   permission:  access/2 child/2
   behavior:  may.sendTo/3 may.getFrom/2 may.return/2
              may.receive/1 may.endow/3
   knowledge:  did.sendTo/3 did.getFrom/3 did.return/2
                did.receive/2 was.endowed/2
system
   /* invoker emits */
   access(A,B) access(A,X) A:may.sendTo(B,X)
    B:may.receive()
    => access(B,X) A:did.sendTo(B,X) B:did.receive(X);

   /* invoker collects */
   access(A,B) access(B,X) A:may.getFrom(B)
    B:may.return(X)
    => access(A,X) A:did.getFrom(B,X) B:did.return(X);

   /* parenthood */
   child(P,C) => access(P,C);
   /* endowment */
   child(P,C) access(P,X) P:may.endow(C,X)
    => C:was.endowed(X);
behavior
   FWDR{
      was.endowed(T) => target(T);
      => may.receive();
      target(T) => may.getFrom(T);
      did.receive(X) child(C) target(T)
       => may.endow(C,X) may.sendTo(T,C);
      did.getFrom(_,X) child(C)
       => may.endow(C,X) may.return(C);}
subject
   alice bob
   proxyAlice:FWDR proxyBob:FWDR
config
   access(alice,alice) access(alice,proxyBob)
   access(bob,bob) access(bob,proxyAlice)
   access(proxyAlice,proxyAlice) access(proxyAlice,alice)
   access(proxyBob,proxyBob) access(proxyBob,bob)
   target(proxyAlice,alice) target(proxyBob,bob)
   child(alice,alice) child(bob,bob)
   child(proxyAlice,proxyBob) child(proxyBob,proxyAlice)
goal
   !access(bob,alice) !access(alice,bob)
```

Figure 6.18: The SCOLL program expressing the safety problem

1. The structure of the language allows developers to describe the different aspects of a problem in six separated parts. It encourages the reuse of common parts and the exploration of the effects of varying every part independently from the other parts.

2. The interdependencies between the different parts are simple, unambiguous, and completely described for the kernel language in section 6.2. They are adapted slightly for the full SCOLL language (Section 6.5).

3. When expressing a programmed entity's behavior restrictions in SCOLL, the most important errors are the ones that compromise the validity of the model as a safe approximation of the effects of interaction between the entities. Therefore, the easiest way to approximate a programmed entity's behavior should be the safest way.

   The easiest way to express behavior in SCOLL is to :

   - either provide behavior rules with empty bodies to express unconditional behavior,
   - or declare the subject without a behavior description to assign the default behavior (see section 6.5.4).

4. Improving the accuracy of the modeled behavior should never lead to an underestimation of an entity's behavior.

   The refinement rules were introduced in SCOLL to meet this objective (Section 6.5.3). Possibilities for further refinement support were introduced and presented as future work in section 6.7.1.

5. The language should allow the user to refine a subject's behavior, without having to adapt the behavior of the other subjects. It should encourage an iterative and incremental approach to behavior refinement, starting from crude safe approximations and refining behavior predicates only when the results fail to indicate that a pattern is safe.

   The refinement rules in SCOLL realize this objective (Section 6.5.3). They avoid the need for adapting the behavior rules of subjects whose behavior does not change, even when new behavior and knowledge predicates are introduced and new system rules are expressed that handle these predicates.

6. The language should provide flexible support for aggregation.

   The example in section 6.9 shows how aggregation can be used in an intuitive way. However, as indicated in sections 6.7.2 and 6.7.3, we think future work in this area may considerably improve SCOLL in this respect. It would greatly enhance the language's potential to perform elaborate experiments if the aggregation relations could be expressed explicitly and the parent-child relations could be expressed independently from the aggregation relations.

Chapter 8 contains several elaborate examples of problems expressed in SCOLL, including their solutions, and gives an indication of the practical usability of SCOLL.

Chapter 7

Pattern Analysis with SCOLLAR

This chapter is an introduction to the SCOLLAR tool. It self contained for all practical purposes, with the exception of the SCOLL syntax that was described in section 6.6. SCOLLAR is a tool for safety analysis that implements the SCOLL language. Where necessary, parts and concepts of the previous chapter are revisited from a practical point of view, often less formal but equally precise and clarified with ready-to-use examples.

Where chapter 6 focussed on the formal aspects of the SCOLL language used to specify patterns of collaborating subjects, this chapter will approach the language and its implementation from the user's point of view and concentrate on how SCOLLAR is to be used and what results can be expected.

The last two sections of this chapter (Sections 7.6 and 7.7) explain the overall design of the tool and its implementation based on Constraint Programming [Sch02] in Mozart/Oz [Moz03].

7.1 Overview

SCOLLAR is a tool to analyze safety in patterns of collaborating entities (subjects) and can be used for several purposes:

1. To check if a given pattern guarantees a set of safety properties without necessarily preventing another set of liveness possibilities.

   For instance, subject `alice` should never get access to subject `bob` (safety) but there should at least be one possible scenario in which `bob` gets access to `alice` (liveness possibility) .

2. To search for (all) safe ways to restrict the interaction between the subjects in the pattern, such that:

   - No safety property is violated
   - No liveness possibility is prevented
   - Every set of restrictions (solution) is minimal for safety: adding a restriction is not necessary, removing a restriction will break at least one safety property.

The user will indicate what restrictions can actually be imposed. Typically, the restrictions affect the behavior of the subjects the user can rely on or control and/or the initial configuration of the pattern.

Using SCOLLAR in this way is most useful when designing and programming secure patterns of interaction between relied upon and some untrusted (unknown) subjects.

The intention and use of both modes will be explained further in section 7.2.

7.1.1 Most Important features

SCOLLAR's most important features are:

1. Behavior and permissions are equally important.

   Authority propagation is not just a matter of how permissions are distributed in the initial configuration (who has the right and the ability to use what in what way) but must also take the behavior of the subjects into account (who is willing to use what permission in what circumstances).

   Both can be specified with equally expressive power.

2. Behavior is based on knowledge:

   Behavior is the intention of a subject to do something. A subject can use its knowledge (about itself and other subjects) to decide to be more collaborative in the propagation of authority. Knowledge is both a prerequisite for and a result of successful authority propagation.

   Subjects describe their intentions as a set of subject rules that generate behavior from their knowledge. The effects of subject interaction that are visible to a subject correspond to knowledge.

3. Subject interaction is mediated explicitly by a system:

   Ultimately, the rules that decide what conditions lead to what effects are the same for every subject. These rules are modeled explicitly as system rules. System rules, like subject rules, are parameterized by subject variables.

   The system rules decide what knowledge can become available to itself and to the subjects. If such a rule is conditional on the behavior of at least two subjects, it is said to be a collaboration rule.

4. Safe but Precise Approximation

   The general problem of precisely calculating if a configuration in a system can lead to the violation of a safety property is not computable. Harrison, Ruzzo and Ullman [HRU76] proved this in 1974 by showing how every Turing machine defines a safety problem that is safe exactly when the Turing machine will come to an halt. In 1936, Turing proved that the halting problem for Turing machines is not computable [Tur37].

   We only consider the maximally possible behavior of the subjects. The subject rules and system rules should be monotonic. More knowledge will make more behavior possible and more behavior can only generate more knowledge. This makes our model a safe approximation of the actual problem.

We will only consider a finite set of subjects, each possibly representing the behavior and knowledge of an infinite number of entities. This technique makes the safety problems in our safe approximation tractable. The formal proof that aggregation results in a safe approximation was given in section 5.7.4.

Within the limits stated above, the approximation can be arbitrary precise. If an approximative model is too crude, one can always refine the system rules to make them generate more precise knowledge from more precise conditions, so that behavior can also be expressed with improved precision.

For instance, a pattern using binary predicate of the form:

```
access(Subject1,Subject2)
```

could add an extra predicate to specify how this access was attained :

```
Subject1:did.getAccessFrom(InvokedSubject,Subject2).
```

This is knowledge informs `Subject1` that it has invoked `InvokedSubject`, who returned `Subject2`.

The latter predicate is denoted in an alternative form, specified in section 5.3.2. We position the first argument in front of the label, followed by a colon. Instead of the normal notation $\mathtt{b(S_1,\ldots,S_n)}$, we write: $\mathtt{S_1:b(S_2,\ldots,S_n)}$.

This notation underlines the fact that it is `Subject1`'s behavior or knowledge we are talking about. The alternative notation is not used for permissions, because permissions are managed only by the protection system.

7.1.2 Restrictions Suggested By SCOLLAR

When looking for solutions that respect the user's safety and liveness requirements, SCOLLAR can suggest two kinds of restrictions to the pattern that was given by the user: behavior restrictions and configuration restrictions.

When searching for minimal sets of restrictions, SCOLLAR will, at least in principle, check all combinations of these restrictions and only report the minimal sets that guarantee the safety and liveness requirements the user has specified. The actual algorithm is smarter and uses constraint programming to avoid having to check every combination separately. It will be described in section 7.6.

The two kinds of restrictions are:

1. Restrictions in the maximal behavior of one or more subjects.

   If the user wants to search for the necessary restrictions in the behavior of a subject, he/she should turn the subject into a search subject. To that end the subject's name must be preceded by the "`?`" sign in the subject part of the pattern description. SCOLLAR will then try to maximize the behavior of all search subjects. Instead of saying that SCOLLAR searches for minimal sets of behavior restrictions, we often also say that SCOLLAR searches for maximal sets of optional behavior facts.

   If the user assigns behavior to a search subject, SCOLLAR will interpret it as a lower bound for the subject's behavior.

2. Restrictions in the initial configuration.

   Every pattern has an initial configuration, which will typically contain initial permissions and/or knowledge for the subjects. To indicate that these initial facts are optional and should be tested by SCOLLAR, they have to be preceded with a "?" sign.

   SCOLLAR will then calculate the maximal subsets of optional configuration facts that do not contradict the user's requirements.

Note that SCOLLAR will merge both kinds of restrictions into one set to be minimized (optional facts to be maximized), because they are inter-dependent. Imposing a restriction of one kind may very well undo the need to impose a restriction of the other kind.

7.2 Different Ways to use SCOLLAR for Safety Analysis

SCOLLAR can be used in two operation modes:

- Fixpoint Computation Mode
- Solutions Mode

7.2.1 Fixpoint Computation Mode

In this mode, SCOLLAR computes the maximal propagation of authority starting from a given configuration. Authority is presented by knowledge facts. Knowledge facts are either subject knowledge facts or system knowledge facts. System knowledge facts represent influence that is exerted on the system and correspond to permissions. Subject knowledge facts represent influence that is exerted on a subject and correspond to the subject's state.

This mode is a simple fixpoint calculation. The system rules and behavior rules are instantiated over the set of subjects and iteratively applied, starting from the knowledge available in the initial configuration until no new facts can be derived. Because the rules are monotonic and the number of derivable facts is finite, such a fixpoint is always reached in polynomial time.

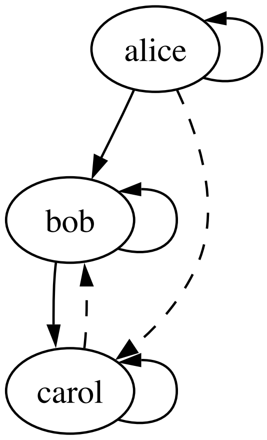
The optional subject behavior facts and configuration facts can be:

- either all disregarded : when computing the "minimal fixpoint"
- or all taken into account : when computing the "maximal fixpoint".

The result is shown on a separate web page (Figure 7.1), containing an access graph of the configuration and a table for every subject.

The access graph is derived from the `access/2` permissions, if such a permission predicate is declared. Otherwise, no graph is shown. The access graph of the initial configuration is presented with solid arcs. The extra access permissions in the fixpoint are presented with dashed arcs.

In the table, reachable facts are presented as `1`, unreachable fact as `0`. The columns represent the subject in the last argument of a fact.



alice	alice	bob	carol
access(alice,_)	**1**	**1**	**1**
alice:did.receive(_)	0	0	0
alice:did.return(_)	**1**	**1**	**1**
alice:did.getFrom(alice,_)	**1**	**1**	**1**
alice:did.getFrom(bob,_)	0	**1**	**1**
alice:did.getFrom(carol,_)	0	**1**	**1**
alice:did.sendTo(alice,_)	**0**	**0**	**0**
alice:did.sendTo(bob,_)	**0**	**0**	**0**
alice:did.sendTo(carol,_)	**0**	**0**	**0**
alice:may.receive()		**1**	
alice:may.getFrom(_)	**1**	**1**	**1**
alice:may.return(_)	**1**	**1**	**1**
alice:may.sendTo(alice,_)	**0**	**0**	**0**
alice:may.sendTo(bob,_)	**0**	**0**	**0**
alice:may.sendTo(carol,_)	**0**	**0**	**0**

bob	alice	bob	carol
access(bob,_)	0	**1**	**1**
bob:did.receive(_)	0	**1**	**1**
bob:did.return(_)	0	**1**	**1**
bob:did.getFrom(alice,_)	0	0	0
bob:did.getFrom(bob,_)	0	**1**	**1**
bob:did.getFrom(carol,_)	0	**1**	**1**
bob:did.sendTo(alice,_)	0	0	0
bob:did.sendTo(bob,_)	0	**1**	**1**
bob:did.sendTo(carol,_)	0	**1**	**1**
bob:may.receive()		**1**	
bob:may.getFrom(_)	**1**	**1**	**1**
bob:may.return(_)	**1**	**1**	**1**
bob:may.sendTo(alice,_)	**1**	**1**	**1**
bob:may.sendTo(bob,_)	**1**	**1**	**1**
bob:may.sendTo(carol,_)	**1**	**1**	**1**

carol			
	alice	bob	carol
access(carol,_)	0	**1**	**1**
carol:did.receive(_)	0	**1**	**1**
carol:did.return(_)	0	**1**	**1**
carol:did.getFrom(alice,_)	0	0	0
carol:did.getFrom(bob,_)	0	**1**	**1**
carol:did.getFrom(carol,_)	0	**1**	**1**
carol:did.sendTo(alice,_)	0	0	0
carol:did.sendTo(bob,_)	0	**1**	**1**
carol:did.sendTo(carol,_)	0	**1**	**1**
carol:may.receive()		**1**	
carol:may.getFrom(_)	**1**	**1**	**1**
carol:may.return(_)	**1**	**1**	**1**
carol:may.sendTo(alice,_)	**1**	**1**	**1**
carol:may.sendTo(bob,_)	**1**	**1**	**1**
carol:may.sendTo(carol,_)	**1**	**1**	**1**

Figure 7.1: The results of calculating a safety problem in SCOLLAR

For instance the fact `alice:did.sendTo(bob,carol)` is presented in the table of `alice`, in the row labeled `alice:did.sendTo(bob,_)` and the column labeled `carol`.

The user's safety and liveness requirements are not taken into account when computing a fixpoint. However, the resulting tables do indicate the cells containing a safety predicate in red and the ones containing a liveness predicate in green.

Results from fixpoint calculations:

1. If the maximal fixpoint is safe and alive (no red cells contain `1` and all green cells do), then all optional facts can be added to the configuration and the behavior, without violating the user's requirements. Consequently, the pattern is safe as it is and no restrictions have to be applied.

2. If the maximal fixpoint is not alive (there is a green cell that does not contain `1`), then there is no solution that does not violate the user's liveness requirements. The required liveness possibilities are prevented by the lack of permissions, initialization knowledge and behavior.

3. If the minimal fixpoint is not safe (there is a red cell that contains `1`), then there is no solution that does not violate the user's safety requirements. The required safety properties are not guaranteed due to the abundance of permissions, initialization knowledge and behavior.

The Fixpoint Computation Mode solves the following problems, defined in section 5.7.2:

- Safety Problems : when only safety properties are required by the user. If the maximal fixpoint violates no safety requirements, the configuration including all optional facts is safe. If the minimal fixpoint violates not safety requirements, the configuration excluding all optional facts is safe.

- Practical Safety Problems : when also liveness possibilities are required by the user. If the maximal fixpoint violates no safety requirements and prevents no liveness possibilities, the configuration including all optional facts is safe and alive. If the minimal fixpoint violates no safety requirements and prevents no liveness possibilities, the configuration excluding all optional facts is safe and alive.

7.2.2 Solution Mode

Often the liveness possibilities will be OK in the maximal fixpoint but not in the minimal fixpoint while the safety properties are OK in the minimal fixpoint but not in the maximal fixpoint. Therefore SCOLLAR offers the solution mode to compute maximal sets of optional facts (minimal sets of necessary restrictions) that provide a solution conform to the user's requirements. SCOLLAR will look for those subsets of the optional facts that at the same time:

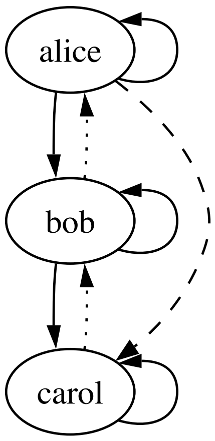
- prevent all the safety properties that are specified by the user,

- do not prevent any of the liveness possibilities that are specified by the user,

- are complete, in the sense that adding another optional fact will break at least one safety property.

All solutions are presented as columns in a table, the rows of which represent the optional facts that are forbidden in at least one solution. A row with all zero's indicates an optional facts that is absolutely forbidden (in every solution, regardless of the other optional facts). Optional facts that are absolutely allowed (in every solution, regardless of the other optional facts) are not shown.

The result is shown on a separate web page (Figure 7.2), containing an access graph and a table.



Solutions	1	2	3	4
alice:may.sendTo(bob,alice)	1	1	**0**	1
alice:may.sendTo(carol,alice)	**0**	**0**	**0**	**0**
alice:may.sendTo(carol,bob)	1	**0**	1	1
bob:may.sendTo(carol,alice)	**0**	**0**	1	1
bob:may.sendTo(carol,bob)	1	**0**	1	1
bob:may.receive()	1	1	1	**0**
bob:may.return((alice)	**0**	1	1	1

Figure 7.2: The results of calculating a maximization problem in SCOLLAR

The access graph is derived from the `access/2` permissions, if such a permission predicate is declared. Otherwise, no graph is shown. The access graph of the initial configuration is presented with solid arcs. The extra access permissions that are common to all solutions are presented as dashed arcs. The extra access permissions that are present in at least one solution are presented as dotted arcs.

The first row of the table contains a button that, when clicked, will show the selected solution in a page similar to figure 7.1.

Every solution can be viewed individually as set of tables, one for every subject, indicating which facts were reached during the computation. This table presents the facts reached concerning a subject as 1, facts that were detected to be unreachable as `0`, and facts that were either not reached or not considered as an underbar character (_).

For the optional facts (behavior and/or knowledge) a fact indicated as _ must be interpreted as not relevant and therefore equivalent to 1, because it does not have an influence on the safety or liveness properties anyway. For the non-optional facts, _ must be interpreted as not reached during the computation, which is of course different from reached (1) as well as from unreachable (0).

The Solution Mode solves the following problems, defined in section 5.7.2:

- Behavior Maximization Problem : when all optional facts are behavior facts, this mode will calculate maximal subsets of these facts that can be generated by the corresponding subject(s) without violating any required safety properties and without preventing any required liveness possibilities.
- Knowledge Maximization Problem : when all optional facts are knowledge facts (initial permissions and subject initialization facts), this mode will calculate maximal subsets of these facts that can be added to the initial configuration without violating any required safety properties and without preventing any required liveness possibilities.
- Configuration Maximization Problem : when the optional facts include knowledge and behavior facts, this mode will calculate maximal subsets of all optional facts, regardless of their nature. In every solution set, the behavior facts can be generated by the corresponding subject's behavior and the knowledge facts can be added to the initial configuration without violating any required safety properties and without preventing any required liveness possibilities.

7.3 Describing SCOLL Patterns

This section describes the basics about how the SCOLL language is used in practice to input patterns in the SCOLLAR tool. We describe here the version of SCOLLAR that very closely corresponds to the version of SCOLL described in chapter 6.

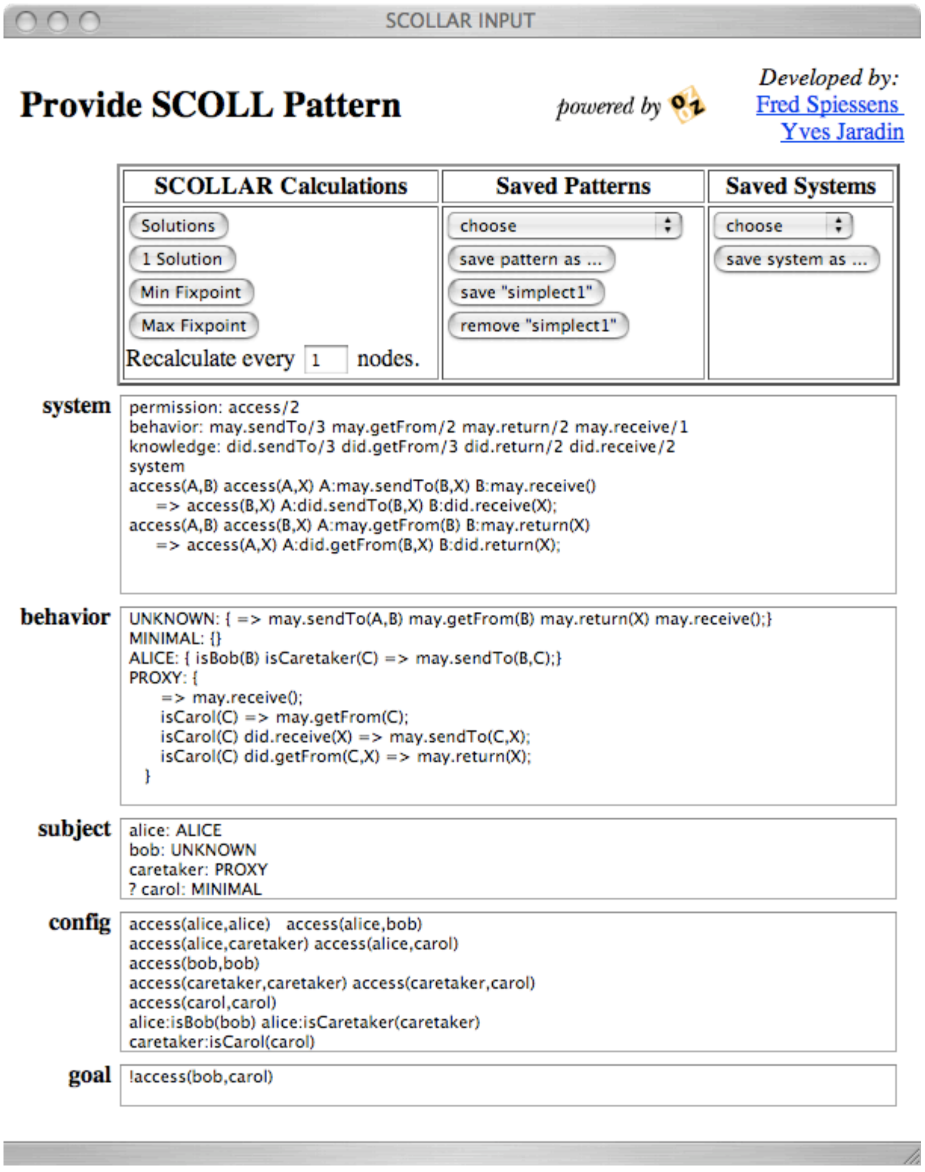
SCOLL programs (also called SCOLL patterns) describe a set of subjects that interact and propagate authority by interacting. The user defines what kind of interaction is considered, what the preconditions for successful interaction are, and what effects it can have.

The complete SCOLL syntax, described in section 6.6, is supported. The six main parts of a SCOLL program are accommodated into five input panes in SCOLLAR's user interface:

1. The system pane : accommodates the `declare` and `system` parts in SCOLL
2. The behavior pane : accommodates the `behavior` part in SCOLL
3. The subject pane : accommodates the `subject` part in SCOLL
4. The config pane : accommodates the `config` part in SCOLL
5. The goal pane : accommodates the `goal` part in SCOLL

The contents of the system pane can be saved separately. Let us first revisit the most important SCOLL concepts from chapter 6, focussing on their practical use, and then explain the practical possibilities offered by the five panes in which a pattern is described.

Figure 7.3 shows an input page for SCOLL patterns in the online version of SCOLLAR in which the five panes can be recognized.



system
```
permission: access/2
behavior: may.sendTo/3 may.getFrom/2 may.return/2 may.receive/1
knowledge: did.sendTo/3 did.getFrom/3 did.return/2 did.receive/2
system
access(A,B) access(A,X) A:may.sendTo(B,X) B:may.receive()
    => access(B,X) A:did.sendTo(B,X) B:did.receive(X);
access(A,B) access(B,X) A:may.getFrom(B) B:may.return(X)
    => access(A,X) A:did.getFrom(B,X) B:did.return(X);
```

behavior
```
UNKNOWN: { => may.sendTo(A,B) may.getFrom(B) may.return(X) may.receive();}
MINIMAL: {}
ALICE: { isBob(B) isCaretaker(C) => may.sendTo(B,C);}
PROXY: {
     => may.receive();
     isCarol(C) => may.getFrom(C);
     isCarol(C) did.receive(X) => may.sendTo(C,X);
     isCarol(C) did.getFrom(C,X) => may.return(X);
  }
```

subject
```
alice: ALICE
bob: UNKNOWN
caretaker: PROXY
? carol: MINIMAL
```

config
```
access(alice,alice)   access(alice,bob)
access(alice,caretaker) access(alice,carol)
access(bob,bob)
access(caretaker,caretaker) access(caretaker,carol)
access(carol,carol)
alice:isBob(bob) alice:isCaretaker(caretaker)
caretaker:isCarol(carol)
```

goal
```
!access(bob,carol)
```

Figure 7.3: SCOLLAR's online user interface: the SCOLL input page

7.3.1 Predicates and Facts

Apart from the subjects themselves, the most central concepts in SCOLL are the relations between the subjects. Every relation that is used corresponds to a predicate with the same arity. A predicate is a Boolean function that indicates if a certain tuple is in the relation.
We denote a predicate as :

```
<label>(A,B,...,X)
```

where `<label>` names the relation and `A ... X` are variables ranging over the subjects. Predicate labels start with a lower case letter, variables with an upper case letter. For instance:

```
read(A,B)
```

is a binary predicate that is true for every pair `(A,B)` such that `A` can read `B`.

In the context of a rule (Section 7.3.3) it is possible to replace a named variable by an underbar character. That will make sure that the variable is unique among all the variables used by the predicates in that rule. For instance:

```
read(A,_)
```

For behavior and knowledge predicates, we use the alternative notation:

`A:label(B)` instead of `label(A,B)`

We call an individual tuple a fact. For instance :

`access(alice,bob)` is a fact. If it is true, it means that `alice` stands in the relation `access` to `bob`.

The first subject in a fact is called the base subject and has a special status: it is the subject whose behavior, knowledge, or permission is expressed by the fact. The role of the base subject is clarified in the next paragraph.

7.3.2 Knowledge and behavior

We distinguish between two kinds of predicates:

Knowledge predicates correspond to relations that are part of the state of the system. We distinguish between permissions and subject knowledge.

Permissions represent system knowledge: the part of the system's state that is used only by the system itself.

For instance, the predicate `access(A,B)` could be used to indicate that a subject A has access to a subject B.

The `permission` keyword in SCOLL indicates the place where the permission predicates are declared.

Subject Knowledge is the part of the system's state that is available to the base subject. It expresses knowledge of the base subject, about its relations with other subjects.

For instance, the predicate `A:did.sendTo(B,X)` could be used to indicate to subject `A` that it has successfully invoked a subject `B` and passed a subject `X` as an input argument in the invocation.

The base subject of a subject knowledge fact is the only subject that can detect it, when the fact becomes available. Subject knowledge is made available to the base subject by the system. Private knowledge is subject knowledge that is generated by the base subject itself.

It is considered good practice to prefix a subject knowledge predicate label with "`did.`", when the knowledge it represents stems from a successful application of the base subject's behavior. The motivation for this prefix is to remind the user of the fact that we are modeling a guaranteed effect of successful behavior.

The `knowledge` keyword in SCOLL indicates the place where the non-private subject knowledge predicates are declared.

Important

In chapters 5 and 6 we used the terms "knowledge" and "knowledge predicate" as general terms to indicate all kinds of knowledge, because it mapped directly to the concept on knowledge in Knowledge Behavior Models. In this chapter we have less use for the general meaning of the term. From now on we will always use "permission" when we mean system knowledge and "knowledge" when we mean subject knowledge.

Behavior predicates correspond to relations that express the intention of the base subject towards the system and towards other subjects. The base subject of a behavior fact is the subject whose behavior is expressed by the fact.

For instance, the predicate `A:may.sendTo(B,X)` could be used to to express that subject `A` is willing to invoke subject `B` and pass access to subject `X` as an input argument in the invocation. It does not imply that `A` is allowed or able to realize its intentions.

It is considered good practice to prefix a behavior predicate label with "`may.`". The motivation for this prefix is to remind the reader of the fact that we are modeling possible behavior.

The following are some guidance rules for modeling relations between entities as predicates in SCOLL:

- To model the right of an entity to change the state of the system in a certain way, use a permission predicate.
- To model the right of an entity to interact (collaborate) with another entity in a certain way, also use a permission predicate.
- To model the internal state of an entity, use a knowledge predicate.
- To express that an entity is willing to perform an action (e.g.: to invoke another entity with a certain input argument), use a behavior predicate.
- To express that an entity is ready to co-operate in an action (e.g.: being invoked with a certain input argument, or returning an output argument), use a behavior predicate.

Examples in the remainder of this chapter will clarify the distinction. Section 7.4.1 will explain how to declare the predicates used in a SCOLL program.

7.3.3 Rules

Rules in SCOLL are a mechanism to derive new facts from existing facts. Every rule has the form of an implication between two conjunctions of predicates.:

```
P ...  Q => R ...  S;
```

where `P ...  R` and `R ...  S` are predicates over a finite number of variables.

Both sides of the implication represent a conjunction of predicates. The left hand side is called the *body* of the rule and can be empty. The right hand side is called the *head* of the rule and always consist of at least one predicate. For instance, this could be a system rule:

```
access(A,B) access(A,C) A:may.introduceTo(B,C)
=> access(B,C) access(C,B) A:did.introduceTo(B,C);
```

During the execution of a SCOLL program, the rules will be instantiated: all variables will be substituted for actual subjects. Both sides of every instance of the rule then represent a conjunction of facts. An instantiated rule is a simple logical implication: all facts in the right hand side will become true if all facts in the left hand side are true.

SCOLL rules never rely on the identity of subjects. Although many programming languages provide some form of identification primitive (like an object equality test), SCOLL requires that the use of such a facility would be modeled explicitly. Therefore SCOLL never allows constant arguments (only variables) in the predicates of the rules.

Counter example:

```
access(A,alice) => access(alice,A); invalid syntax
```

Good example:

```
isAlice(X) access(A,X) => access(X,A);
```

In a future version, SCOLL may provide special binary predicates for this purpose and allow the use of constants in these two special predicates only:

```
=(X,Y)
≠(X,Y)
```

7.4 The Distinct Parts of a SCOLL Pattern

This section describes how the input panes in SCOLLAR are used and how they relate to the corresponding parts of a SCOLL pattern.

7.4.1 The system pane

This pane combines the two first parts in a SCOLL program:

1. the `declare` part, in which the predicates are declared

2. the actual `system` part, in which the system rules are given

The keyword `declare`, that indicates the start of the predicate declaration parts of the SCOLL program, is dropped here, but the keyword `system` must be used to indicate where the system rules start. The `declare` keyword will be inserted at its proper place, before the SCOLL program is handed to the parser.

The reason these two parts are combined into one pane is administrative convenience. The combination can easily be reused in other patterns and can be saved separately from the rest of the pattern. Both parts may be separated again in a future version of SCOLLAR.

First, the predicates are declared in their proper section (`permission`, `behavior`, or `knowledge`, as:
`<predicateLabel>`, followed by "/" and a strict positive number that indicates the number of arguments (arity) the predicate takes.

For instance:

```
permission :   access/2

behavior :   may.sendTo/3 may.getFrom/2 may.return/2
     may.receive/1

knowledge :   did.sendTo/3 did.getFrom/3 did.return/2
     did.receive/2
```

The order in which the three predicate declaration sections appear is fixed. Within every section the order of the defined predicates is arbitrary and has no influence on the semantics of the program.

Following the predicate declaration, the `system` keyword is added and the system part is described as a list of system rules. A system rule can have one of four types:

permission rule : The normal type of system rule that has permission predicates and/or behavior predicates in its body and permission predicates and/or knowledge predicates in its head. Each permission rule models a certain type of (inter-) action: the conditions that are necessary (permissions and behavior) and the effects that will result from it (permissions and knowledge).

Example:
```
access(A,B) access(B,Y)
A:may.getFrom(B) B:may.return(Y)
=> access(A,Y) A:did.getFrom(B,Y) B:did.return(Y);
```

`A`'s knowledge `A:did.getFrom(B,Y)` lets `A` be aware of its successful collaboration with `B` and of the fact that `A` gained access to `Y` from that collaboration. `B`'s knowledge `B:did.return(Y)` tells `B` that it has returned `Y`, but it does not tell `B` who it returned `Y` to. The example models invocation (of `B` by `A`) in systems that do not reveal the invoker to the invoked entity, a situation that is not uncommon in programming languages.

assisting rule: A rule that extends the effects of one or more other rules. It has one or more knowledge predicates and/or permissions in its body and one or more permissions and/or knowledge predicates in its head.

Example:
`A:did.getFrom(B,Y) => access(A,Y) B:did.return(Y);`

Assisting rules should be used with caution. As they look very much like the actual permission rules, they can easily cause confusion about the the actual propagation mechanisms that are modeled.

knowledge refinement rule: A rule that expresses how knowledge predicates relate to their refinements. Its body is a single refined knowledge predicate and its head contains the unrefined knowledge predicate that is implied by it. Such a rule is useful to support the refinement of an existing pattern that did not use the refined knowledge before.

Example:
```
B:did.returnFor(_,Y) => B:did.return(Y);
B:did.returnFor0(Y) => B:did.return(Y);
```
The first rule can be used to express the knowledge that `B` has returned `Y` after making sure that some other subject was provided as input to `B` in the same invocation (`B:did.returnFor(_,Y)`), implies the simpler knowledge that `B` has returned `Y`.

The second rule can be used to derive the same general knowledge from the specific knowledge that nothing was provided as input to `B`, when it returned `Y`.

behavior refinement rule: A rule that expresses how behavior predicates relate to their refinements. Its body is a single unrefined behavior knowledge predicate and its head contains the refined behavior predicates that are implied by it. Such a rule is useful to support the refinement of an existing pattern that did not use the refined behavior before.

Example:
```
B:may.return(Y)
=> B:may.returnFor0(Y) B:may.returnFor(_ Y);
```
This rule can be used to express that if `B` is willing to return `Y` regardless of whether some input is provided in its invocation (unrefined behavior), then B is also willing to return `Y` in the specific circumstances when such input is (or is not) provided.

7.4.2 The behavior pane

The behavior pane (American English spelling) will contain a list of zero or more behavior declarations. A behavior declaration consists of a behavior identifier and a list of zero or more subject rules and has this form:

```
<BEHAVIORNAME> { <rule> ... <rule>}
```

The behavior name is a string of all uppercase letters. The subject rules have the same format as the system rules in section 7.4.1: they are implications between conjunctions of predicates.

The body of a subject rule contains zero or more knowledge predicates. The conjunction of these predicates is the precondition for the subject rule to expose behavior.

The head of a subject rule contains one or more behavior predicates and/ or **private knowledge predicates**. The behavior predicates describe what behavior will be exposed if the preconditions of the rule are satisfied for a given set of subjects.

For instance, here is a simple behavior class:

```
DISPATCHER {

=> may.receive();

did.receive(X) => may.return(X);}
```

Important

Notice that the predicates `may.receive`, `did.receive` and `may.return` have one less argument than was indicated by their declaration in the example of section 7.4.1. The first argument is **always dropped** in subject rules, for reasons which will be explained soon.

Subject rules are a monotonic approximation of how subjects of a certain behavior class decide under what circumstances they want to collaborate in what way. Monotonic means that when more knowledge (preconditions) is available no less behavior will be generated by the rules. Once a behavior fact has been set to true (derived), it cannot be set to false again, because the knowledge necessary to generate the behavior will never disappear.

Private Knowledge

Private knowledge is knowledge that is generated by the base subject, or knowledge that was given to the subject upon initialization. Private knowledge can also be useful to combine knowledge the subject has learned during interactions and to represent the subject's internal "state".

Private knowledge predicates are automatically declared the first time they are used in a behavior declaration. Their scope extends to every behavior rule within that declaration. Upon their first appearance in a behavior declaration, every predicate that is used in a subject rule and that was not declared in the system pane (`declare` part), will be declared automatically as a private knowledge predicate. It will be given the arity that can be derived from the number of variables used in that occurrence, *plus one* to account for the implicit base subject variable.

Why the base subject is implicit in behavior rules.

Subject rules can only use knowledge that is available to the base subject. They specify the behavior of the base subject. Instead of demanding that every predicate in a subject rule should have the same base-subject variable, we simply drop the first argument in the predicate (make it implicit).

However, the reason for making the base-subject implicit is not just simplification. To understand the importance of implicit base-subjects, consider the following counter example of a subject rule in which the first argument is explicit:

Counter example subject rule :

```
=> A:likes(A);
```

This rule would mean that every subject of this behavior class likes itself. Using this rule makes the following implicit assumptions :

- The real system that is being modeled in SCOLL provides and infallible identification facility that allows every subject (or at least every subject of this behavior class) to differentiate between itself and other subjects.
- The subjects modeled in this behavior always (!) use this identification before making any decision.

These assumptions are very strong, particularly the second one, and do not hold in most general. If the first assumption does hold, it should be expressed explicitly in the system rules, for instance by a system rule like :

```
=> self(A,A);
```

If the second assumption also holds, the use of the identification knowledge in a subject rule should be explicit, for instance:

```
self(A) => likes(A);
```

Example of a behavior:

```
FORWARDER { => may.receive();
            did.receive(X) target(T) => may.sendTo(T,X);}
MINIMAL {}
```

`FORWARDER` is the behavior for subjects that are willing to receive subjects (e.g.: be invoked with an input argument) and forward them to their target subject. The example assumes that `did.receive/2` was declared as a knowledge predicate, that `may.sendTo/3` was declared as behavior predicate, and that `target/2` is a private knowledge predicate that is automatically declared here. It also assumes that the subjects with `FORWARDER` behavior will be initialized with knowledge about who their target(s) is (are).

`MINIMAL` is the behavior for subjects that never collaborate.

DEFAULT behavior

If the name of a behavior class is "`DEFAULT`", all subjects declared in the subject pane that have no behavior assigned explicitly will have this behavior. If the behavior class `DEFAULT` is not specified, the default behavior is defined with a single behavior rule with an empty body and a head that contains all declared behavior predicates :

DEFAULT { => behavior$_1$(_,...,_) ... behavior$_n$(_,...,_);}

7.4.3 The subject pane

The subject pane is to contain a list of subject declarations of the form:

```
<subjectName> :  <BEHAVIORNAME>
```

or, if the subject has `DEFAULT` behavior:

```
<subjectName>
```

The subject name starts with a lower case letter and identifies the subject. The behavior name identifies the subject's behavior and must be declared in the behavior pane.

Behavior Maximization Option : "?"

A question mark ("?") can precede the subject name to indicate that SCOLLAR should maximize the behavior of this subject, using the specified behavior as a lower bound. Every possible behavior fact which base subject is the declared subject becomes an optional behavior fact.

Example :

```
alice :  FORWARDER
bob
?carol:  MINIMAL
```

Subject `carol` is marked with the `search` flag. That indicates that SCOLLAR should find solutions that maximize `carol`'s behavior (in excess of `carol`'s empty `MINIMAL` behavior) while respecting the safety and liveness constraints in the config pane.

7.4.4 The config pane

The config pane is to contain a list of permission facts, knowledge facts and private knowledge facts that represent the initial state of the configuration when the system rules and behavior rules start deriving new behavior and knowledge.

```
<fact> <fact> <fact> ...
```

All facts must be instantiations of predicates that are declared, either explicitly in the system pane, or automatically (as private knowledge) in the behavior declaration of their base subject. Their arguments must all be subjects that were declared in the subject pane.

Example :

```
access(bob,alice) access(alice,carol)
access(bob,bob) access(alice,alice)
access(carol,carol) alice:target(carol)
```

Configuration Maximization Option

Every configuration fact can be preceded by the "?" mark that turns it into an optional fact. SCOLLAR will search for solutions that maximally include these optional facts.

Example :

```
access(bob,alice) access(alice,carol)
access(bob,bob) access(alice,alice)
access(carol,carol) alice:target(carol)
?access(carol,alice) ?alice:target(bob)
```

The search option in the config pane can be used in combination with the search option in the subject pane. SCOLLAR will search for solutions that maximally set all optional facts to true.

7.4.5 The goal pane

The goal pane is to contain a list of:

- permission, knowledge and private knowledge facts that represent the liveness possibilities
- permission, knowledge and private knowledge facts preceded by "`!`" that represent the safety properties

When SCOLLAR is searching for solutions it will only report solutions in which all liveness possibilities are reached and no safety property is reachable.
Example :

```
access(carol,bob) /* liveness possibility */
!access(bob,carol) /* safety property */
```

SCOLLAR will search for solutions that guarantee that `bob` will never have access to `carol`, while `carol` can get `access` to `bob`, at least if every subject always uses its maximal behavior.

IMPORTANT:

SCOLL programs do not model exact behavior, but safe (over-)approximations of behavior. Safety properties guaranteed in the approximation will be guaranteed in the real problem too. However, liveness properties cannot be guaranteed at all, since the modeled behavior will usually be a strict over-approximation of the actual behavior.

The only use of liveness possibilities, is to require a lower bound to the maximally possible generation of facts in the pattern.

7.5 SCOLLAR's Web Based User Interface

SCOLLAR has a browser based user interface. The main page contains the five text panes that were described in section 7.4 and that allow the user to type the corresponding six parts of the SCOLL pattern (see Chapter 6). This section explains the remaining interface elements.

7.5.1 SCOLLAR Calculations

This button starts a calculation in the first mode : to compute the set of all facts that are reachable from the configuration. This set represents an upper bound for the propagation of authority in the actual problem that was modeled as a SCOLL pattern.

The optional facts specified in the subject pane or in the config pane are not considered to be part of the configuration.

The result is shown on a separate web page (Figure 7.1), containing an access graph of the configuration and a table for every subject.

The access graph is derived from the `access/2` permissions, if such a permission predicate is declared. Otherwise, no graph is shown. The access graph of the initial configuration is presented with solid arcs. The extra access permissions in the fixpoint are presented as dashed arcs.

Every table presents the reachable facts concerning one subject with `1` and the unreachable facts as `0`. The columns represent the subject in the last argument of a fact. For instance the fact `alice:did.sendTo(bob,carol)` is presented in `alice`'s table, in the row labeled `alice:did.sendTo(bob,_)` and the column labeled `carol`.

Max Fixpoint

This button starts a calculation in the first mode : to compute the set of all facts that are reachable from the configuration. This set corresponds to the maximal propagation of authority in the actual problem that was modeled as a SCOLL pattern.

This time, the optional facts specified in the subject pane and in the config pane are considered to be part of the configuration.

The result is shown on a separate web page, similar to the one for the minimal fixpoint calculation (Figure 7.1).

Solutions

This button causes SCOLLAR to start calculating all solutions it can find in a predefined time (by default: 30 seconds). SCOLLAR is then used in the second operation mode to find those sets of optional facts at the same time:

- prevent all the safety properties that are specified in the goal pane,
- do not prevent any of the liveness possibilities that are specified in the goal pane,
- are complete, in the sense that adding another optional fact will break at least one safety property.

Every solution will list a minimal set of restrictions: optional behavior facts and optional facts in the initial configuration that have to be prevented to guarantee the safety properties. Only the solutions that do not prevent the liveness possibilities are listed.

The result is shown on a separate web page (Figure 7.2), containing an access graph and a table.

The access graph is derived from the `access/2` permissions, if such a permission predicate is declared. Otherwise, no graph is shown. The access graph of the initial configuration is presented with solid arcs. The extra access permissions that are common to all solutions are presented as dashed arcs. The extra access permissions that are present in at least one solution are presented as dotted arcs.

The solutions are presented in the columns of the table. Every optional fact that is to be prevented in at least one of the solutions is presented in a row. The optional facts that have to be prevented in a solution have a zero in the corresponding cell. The optional facts that have to be prevented in all solutions can easily be recognized as their row consists only of zero's.

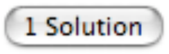
The first row of the table contains a button that, when clicked, will show the selected solution in a page similar to figure 7.1.

Every solution can be viewed individually as set of tables, one for every subject, indicating the facts as follows:

0 : The optional facts that are not allowed and the facts that were detected to be definitely unreachable during the calculation.

1 : The facts that are reached during the calculation and the optional facts that were relevant to the calculation and turned out to be allowed.

_ : The optional facts that are allowed but were irrelevant to the calculation and the facts that were not reached, either because they were not relevant in the calculation, or because they were unreachable.



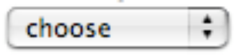
This button is similar to the previous one, but the calculation stops as soon as a first solution if found.

7.5.2 Saved Patterns

This part of the user interface allows the user to manage examples of complete patterns. You can only save patterns if your browser's cookies are enabled. The cookie SCOLLAR creates is called "storageKey" and has a random value assigned to it.

You can only delete or update the patterns you saved yourself with that cookie's value. You cannot change or save predefined example patterns, but if you try to, a personal (private) copy will be made of that pattern.

The availability of the buttons in this section depends on the context and on whether you have cookies enabled in your browser.



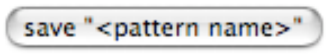
This menu button provides some instructive examples of complete SCOLL patterns. Choose option "empty" to clear all fields of the current pattern.

The patterns you saved while using your current cookie "storageKey" are also available, recognizable by the "(*)" prefix. In the current version you cannot use patterns you saved with another value of the "storageKey" cookie.

This button allows you to save the current pattern under a different name.
Use only alpha-numeric characters for the name of the pattern. Do not use "(*)" to indicate your own patterns. That information will be derived automatically.

If you use the name of a pattern you saved previously, it will be overwritten without warning.

If you use the name of a predefined example pattern, your own instance of a pattern with that name will be overwritten, or one will be created if it does not exist.



"<pattern name>" will contain the name of the pattern you last loaded or saved

This button allows you to save changes to the pattern you last loaded or saved, under the same name.

If the current pattern is a predefined example pattern, your own instance of a pattern with that name will be overwritten, or one will be created if it does not exist.

"<pattern name>" will contain the name of the pattern you last loaded or saved

This button allows you to remove the pattern you last loaded or saved.

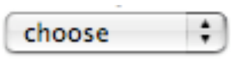
If the current pattern is a predefined example pattern, your own instance of a pattern with that name will be removed instead, or nothing will happen if it did not exist.

7.5.3 Saved Systems

This part of the user interface allows the user to manage examples of reusable system panes for patterns.. You can only save systems if your browser's cookies are enabled. The cookie SCOLLAR creates is called "storageKey" and has a random value assigned to it. The same cookie is used for saving complete patterns.

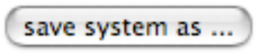
You can only delete or update the system panes you saved yourself with that cookie's value. You cannot change or save predefined example system panes, but if you try to, a personal (private) copy will be made of that system pane.

The availability of the buttons in this section depends on the context and on whether you have cookies enabled in your browser.



This menu button provides some instructive examples of reusable system panes for SCOLL patterns.
Choose option "empty" to clear the system field of the current pattern.

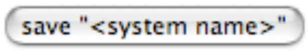
The system panes you saved while using your current cookie "storageKey" are also available, recognizable by the "(*)" prefix. In the current version you cannot use system panes you saved with another value of the "storageKey" cookie.



This button allows you to save the current system pane under a different name.
Use only alpha-numeric characters for the name of the pattern. Do not use "(*)" to indicate your own system panes. That information will be derived automatically.

If you use the name of a system pane you saved previously, it will be overwritten without warning.

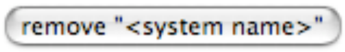
If you use the name of a predefined example system pane, your own instance of a system pane with that name will be overwritten, or one will be created if it does not exist.



"<system name>" will contain the name of the system pane you last loaded or saved.

This button allows you to save changes to the system pane you last loaded or saved, under the same name.

If the current system pane is a predefined example, your own instance of a system pane with that name will be overwritten, or one will be created if it does not exist.



"<system name>" will contain the name of the system pane you last loaded or saved.

This button allows you to remove the system pane you last loaded or saved.

If the current system pane is a predefined example, your own instance of a system pane with that name will be removed instead, or nothing will happen if it did not exist.

7.6 Overall CCP-based design

We consider a single constraint store consisting of elementary boolean constraints on a finite set of variables, whereby the constraint store represents the conjunction of all these constraints. Every variable corresponds to either a permission fact, a knowledge fact, or a behavior fact in the (kernel) SCOLL program. In figure 7.4, the store is represented by the three grayed ellipses.

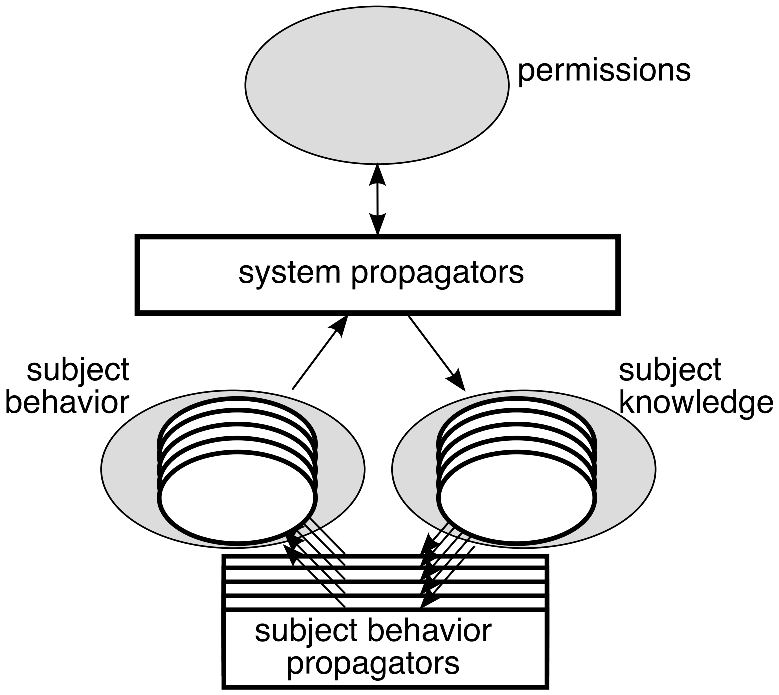


Figure 7.4: Constraint Store and Propagators

The store contains a variable for every permission fact, knowledge fact, and behavior fact that is defined by the set of declared predicates over the set of declared subjects.

The following two basic operations are defined on a constraint store:

$ask(C)$ **:** The operation compares the basic constraint C with the constraint store and blocks until the store is constrained enough to either imply C or imply $\neg C$.

$tell(C)$ **:** The operation adds the basic constraint C to the constraint store, unless that would cause the store to contain contradictory information. In the latter case the store becomes *failed*.

From these basic operations, *constraint propagators* can be constructed. Constraint propagators consist of two parts:

1. A (basic or non-basic) constraint that will be compared with (*asked to*) the constraint store. The propagator will block until the constraint is entailed or disentailed by the store.

2. A (basic or non-basic) constraint that will be added (*told*) to the constraint store if and when the first constraint is entailed by the store. If and when the first constraint is disentailed by the store, nothing happens.

In Concurrent Constraint Programming (CCP) all propagators are concurrent: they block until their preconditions are entailed or disentailed, without blocking any other constraints. The execution order of the constraints is not important, because CCP is confluent: the constraint store will always reach the same final state, regardless of the execution order. For a formal explanation of CCP and its semantics, see [SRP91] or [FA03]. How constraints are used in Mozart/Oz is explained in [Sch02].

The variables corresponding to the non-optional facts in the `config` part of the SCOLL program are constrained to be equal to *true*. The system rules and the subject rules are instantiated over all subjects and transformed into constraint propagators that *ask* for every variable that corresponds to a fact in their body, whether that variable is constrained to be *true*. If and when all these variables are indeed constrained to be true, the propagator will *tell* the constraint (`V` = *true*) to the store, where `V` is the variable that corresponds to the fact in the instantiated rule's head.

As shown in figure 7.4, the system propagators will *ask* constraints requiring variables that correspond to subject behavior and permissions to be constrained to *true* (incoming arrows) and *tell* constraints that set variables corresponding permissions and subject knowledge to be *true*. The subject propagators only *ask true*-constraints in the part of the store that corresponds to knowledge facts about that subject (the base subject of the knowledge fact). The subject propagators only *tell true*-constraints in the part of the store that corresponds to behavior facts about that subject (the base subject of the behavior fact).

Allowing all propagators to run until the store becomes stable, we calculate the fix points of the SCOLL program (SCOLLAR's first operation mode).

To calculate a maximal set of optional facts (SCOLLAR's second operation mode), we tell basic constraints for every safety property, constraining their corresponding variable to be *false*. We then run the propagators starting from a configuration with no optional facts, wait for the store to become stable, and tell constraints for the optional facts one by one, constraining them to be true, and waiting until the store becomes stable before telling the next one.

If instead of becoming stable, the store becomes *failed*, we constrain the last added optional fact to be *false* instead, and we try another one. The store can become *failed* when a system rule propagator *tell*s a variable corresponding to a safety property to be *true*.

If the store is still stable after all optional facts have been constrained this way (to either *true* or *false*), we check if all the variables that correspond to liveness possibilities have been constrained to be *true* in the store. If so we have a solution. If not we have to try another combination of truth values for the optional facts. A solution will automatically be maximal, because we always try to constrain the optional facts to be *true* first, before trying to constrain it to *false*.

Because we may have to try all these combinations before finding a solution, the maximization problems are NP-complete. This claim will be proved in chapter 9. To improve the performance, more propagators will be added to prune the search space consisting of the truth value combinations of all optional facts and smarter search strategies will be used.

7.6.1 Propagation

The propagators can be derived directly from the system rules and behavior rules in the kernel SCOLL program. A system rule of the form:

```
pred$_1$(V$_{1,1}$,...,V$_{1,k}$) ... pred$_n$(V$_{n,1}$,...,V$_{n,m}$)
=> pred$_{n+1}$(V$_{n+1,1}$,...,V$_{n+1,l}$)
```

will be instantiated for every possible substitution that maps the variables in the rule to subjects, replacing the variables in the rule by the corresponding subject in the substitution. For example, the propagator corresponding to a particular instantiation may look like:

$$\frac{pred_1(alice,\ldots,bob) \land \ldots \land pred_n(bob,\ldots,carol)}{pred_{n+1}(alice,\ldots,carol)} \tag{7.1}$$

In (7.1) the *ask* part of the propagator is presented on top of the *tell* part.

The behavior rules are translated in a similar way, but the substitution is somewhat more involved. After substituting the variables in the rule, the predicates will have to be extended with an extra first argument that corresponds to the subject whose behavior is described by the rule.

For instance the unrestricted behavior of a subject `dave`, described by the SCOLL subject rule:

```
=> pred$_1$(...) ... pred$_n$(...)
```

will, after having translation of the rule to kernel SCOLL, and after proper substitution, correspond to a series of propagators of the form:

$$\frac{true}{pred_1(dave,\ldots)} \cdots \frac{true}{pred_n(dave,\ldots)} \tag{7.2}$$

7.6.2 Declarative Laziness

When instantiating all rules over all possible subjects substitutions, the number of variables in the core has the order of magnitude $\mathcal{O}(s^a)$ where s is the number of subjects in the SCOLL program and a is the maximum arity of all predicates. The number of propagators has the same order of magnitude.

Many of the instantiated variables and propagators may never be used, or may not be relevant to the solution of the safety problem or the maximization problem. To minimize the overhead, the instantiation of the rules and the variables is done lazily (on demand).

The Mozart implementation of Oz provides a `ByNeed` operation that was adapted to be confluent, as we reported in [SCR03]. A confluent `ByNeed` can be very useful in declarative paradigms like constraint programming. We use it to generate constraint propagators on demand, only if and when they are needed.

Every variable in the constraint store corresponds to a fact `pred(...)`. When a boolean constraint about that variable in the constraint store is *ask*ed by a propagator, all the rules that have the `pred(...)` predicate in their head will be instantiated, but only for those substitutions that correspond to the fact.

For instance, consider a SCOLL pattern with four subjects `alice`, `bob`, `carol`, and `dave`, and the two following system rules generate `access()`:

```
access(A,B) access(A,X) A:may.sendTo(B,X) B:may.receive()
=> access(B,X)

access(A,B) access(B,Y) A:may.getFrom(B) B:may.return(Y)
=> access(A,Y)
```

When a boolean constraint is *ask*ed about the variable that corresponds to the fact `access(a,c)`, both rules will be partially instantiated to generate two instances of each of the following propagators:

$$\frac{access(s_1,a) \wedge access(s_1,c) \wedge may.sendTo(s_1,a,c) \wedge may.receive(c)}{access(a,c)} \tag{7.3}$$

$$\frac{access(a,s_1) \wedge access(s_1,c) \wedge may.getFrom(a,s_1) \wedge may.return(s_1,c)}{access(a,c)} \tag{7.4}$$

where S_1 ranges over $\{bob, dave\}$, a denotes $alice$ and c denotes $carol$.

To further diminish overhead, the boolean preconditions of a propagator will be *ask*ed one by one. For instance the propagator

$$\frac{access(bob,a) \wedge access(bob,c) \wedge may.sendTo(bob,a,c) \wedge may.receive(c)}{access(a,c)}$$

will first wait for the variable *access(bob,alice)* to be constrained. When that variable becomes bound to *true*, it is will *ask* for the next one: *access(bob,carol)*, but if it becomes bound to *false*, the propagator's work is done, and no other constraint store variables need to be *ask*ed.

The process of instantiating system rules and behavior rules and transforming them to propagators is started by *ask*ing for the constraint store variables that correspond to the facts in the goal part of the SCOLL program.

During the search process to solve a maximization problem, every time an optional fact is bound to *false*, it will also be `ask`ed to make sure that when a SCOLL rule derives this fact to be *true*, the constraint store will *fail*.

In fix point mode, *all* facts will be asked regardless of the overhead, because the truth value of all facts must be derived, not only of the facts that can be relevant to infer the truth value of a goal fact. Because the computation is done only once in fix point mode, the overhead is not that important.

7.6.3 Closed World Propagators

Consider a maximization problem in a SCOLL pattern with subjects {`alice, bob, carol, dave`}, where the propagators that can *tell* $access(alice, carol)$ are defined as in (7.3) and (7.4), for $s_1 \in \{bob, dave\}$. Only these four propagators can cause the variable *access(alice, carol)* to become *true*.

If all four preconditions of these propagators become constrained to be false and `access(alice carol)` is not an initial fact in the `config` part of the SCOLL program, no other propagator can cause *access(alice, carol)* to be come `true`. In that case, the constraint store entails that *access(alice, carol)* is *false*.

Upon instantiating the four propagators that can *tell* $access(alice, carol) = false$, in case `access(alice carol)` is *not* in the `config` part, a fifth propagator will be created that *ask*s if the disjunction of these four conditions is *false* and then *tell*s: *access(alice,carol) = false*.

This extra propagator closes the world of possibilities for the fact to become true. That is useful because it can cause early detection of failure by *tell*ing liveness possibilities to become false.

Because of the closed world propagators, during the search process to solve a maximization problem, the optional facts will no longer only be *ask*ed when they are bound to *false*, but also when they are bound to *true* to make sure that, when a closed world rule derives this fact to be *false*, the constraint store will *fail*.

The use of closed world propagators does not guarantee that the store will either fail or be completely bound. Variables that are asked may remain unconstrained when neither the rule-generated propagators nor the closed world propagator can *tell*.

Even when using closed world propagators and all optional facts have been assigned a truth value and the store is stable, the store is still not guaranteed to represent a solution, because a liveness property may not be constrained to either *true* or *false*. The store only contains a solution if all liveness properties are constrained to `true`.

7.6.4 Distribution

When searching for maximal sets of optional facts that can be constrained to `true` without violating the safety properties (without making the store fail), we assign a truth value to them, one by one and wait for the store to become stable.

In this process, two phases are alternated:

Propagation : The propagators tell constraints until the store is stable.

Distribution : A constraint is added to the store, about an optional fact that is not yet constrained in the store.

The propagation phase was explained in section 7.6.1. During the distribution phase, two decisions must to be made:

1. What optional fact will be constrained next.

2. How will the fact be constrained.

To improve the chances of early detection of failure, we will first constrain the optional facts that were `ask`ed for by the largest number of propagators. To make sure that we find a maximal solution, we always constrain the fact to be *true* first. If no (more) solution(s) are found where this fact is `true`, only then will we try to constraint the fact to be *false* instead.

7.6.5 Search

Propagation and distribution take place in a computation space [Sch02]. A computation space corresponds to a version of the constraint store and the propagators, after a propagation phase has ended and before the distribution phase begins.

In the Mozart [Moz03] implementation of Oz [Smo95, VH04], computation spaces are first class citizens of the programming language. Before every distribution step, a copy of the computation space is set aside. When the store fails in the current computation space, or no more solutions can be found with the chosen optional fact bound to *true*, the computation will resume in the copy, but now with the optional fact bound to *false*.

When a store is still stable when all optional facts have been assigned a truth value, this combination of truth values is set aside in a special list that can be accessed by all computation spaces.

Before a computation is resumed in the copy of a computation space, this list of previously found solutions is consulted and an extra propagator is added that will constrain the set of optional facts are bound to *true* to be *no subset* of any of the solutions already found.

This is a branch and bound strategy that guarantees that all our solutions are maximal and also improves the performance of the search by pruning the search space of valid combinations of truth assignments of optional facts.

7.7 Implementation

7.7.1 Using Finite Domain Integers

The current implementation of SCOLLAR is based on finite domain integers constraints. Every permission, knowledge, and behavior fact is represented as a finite domain integer variable with a domain ranging from 0 (*false*) to 1 (*true*). Logical connectives can be implemented as a product constraint (logical and) or a sum constraint (logical or).

For instance, the branch and bound propagator mentioned in section 7.6.5 is expressed as a constraint on the sum of the finite domain integer variables that were 0 (*false*) in a previously found solution. As the new set of variables bound to 1 cannot be a subset of an earlier one, at least one variable that was bound to 0 in the previous solution must now be bond to 1.

Figure 7.5 shows the Oz code for this propagator.

```
proc{MoreSolutions Old New}
      % all further solutions must have at least one zero
      % replaced by a one  to avoid being a sub solution
      % this goes even if the current solution is not alive.
   (New.oldTraces) := {ZeroPredSpec Old}|@(Old.oldTraces}
   {ForAll @(New.oldTraces)
     proc{$ OldZeroPreds}
        {FD.sum {Map OldZeroPreds fun{$ Pred} {GetPred Pred
New} end}
          ´>:´ 0}
   end}
end
```

Figure 7.5: The Oz finite domain integer implementation of a constraint propagator that guarantees that only maximal solutions are found

The procedure `MoreSolutions` installs the finite domain constraint propagator `FD.sum` for every previous solution `Old`, to make sure that at least one variable in the

`New` solution that corresponds to a fact that was bound to 0 in the `Old` computation space, will now be bound to 1.

The procedure `MoreSolutions` will be called whenever a solution is found, even when this solution did not infer the liveness possibilities as required by the SCOLL program. In that case, it would be futile to look for solutions with a smaller set of optional facts bound to 1.

The idea for this approach was provided to us by Raphaël Collet.

7.7.2 Alternative Approach using Finite Sets

An alternative approach, based on finite set constraints, was proposed and worked out by Yves Jaradin [SJV05], but was not yet integrated into SCOLLAR. Instead of individual facts that describe a relation between n subjects, the finite set variables represent n-tuples of subjects that satisfy a predicate.

A unique integer is assigned to each n-tuple of subjects to represent that tuple in the set. Implications over predicates are translated into set inclusions over the corresponding finite sets of integers. Disjunction is translated to union and conjunction becomes intersection.

Because these operations are only valid when they are applied on compatible finite set representations of predicates, some finite set representations of the predicates may need to be adjusted. Consider the following behavior declaration:

```
FORWARDER {
    => may.receive();
    did.receive(X) target(T) => may.sendTo(T,X);}
```

Remember: the predicates in subject rules have an implicit first argument. Their actual arity is one more than the number of arguments shown in the subject rules.

A finite set variable $may.receive$ will represent the set of subjects s_1 such that the fact $may.receive(s_1)$ is *true*. Two finite set variables will represent sets of pairs of subjects: $did.receive$ contains the pairs (s_1, s_2) such that $did.receive(r_1, r_2)$ is *true* and $target$ contains the pairs (s_1, s_2) such that $target(s_1, s_2)$ is `true`. Another finite set variable $may.sendTo$ will represent the set of triplets of subjects (s_1, s_2, s_3) such that the fact $may.sendTo(s_1, s_2, s_3)$ is *true*. Finally, the finite set `subject` contains all subjects in the SCOLL pattern.

To find the triplets of subjects (s_1, s_2, s_3) such that $did.receive(s_1, s_2)$ is *true* and $target(s_1, s_3)$ is *true*, we cannot simply take the intersection of the *did.receive* and *target* finite set variables. First we have to make cartesian products and permutations in the following way:

The finite set $\{(s_1, s_2, s_3)|did.receive(s_1, s_2)\} = did.receive \times subject$

The finite set $\{(s_1, s_2, s_3)|target(s_1, s_2)\} = target \times subject$

The finite set $\{(s_1, s_2, s_3)|target(s_1, s_3)\}$ is the permutation $P_{2,3}(target \times subject)$

The finite set $\{(s_1, s_2, s_3)|did.receive(s_1, s_2) \wedge target(s_1, s_3)\} = (did.receive \times subject) \cap P_{2,3}(target \times subject))$

The cartesian product and the permutations of the finite sets are implemented by recalculating the individual integers that represent the tuples of the result set.

To calculate a propagator for a rule that has less variables in its head than in its body, one more kind of operation is needed: the element-wise projection onto sub-tuples. This is also implemented by recalculating the integers that represent the tuples.

The second rule thus translates to the finite set constraint:

$$may.return \subseteq (did.receive \times subject) \cap P_{2,3}(target \times subject)).$$

All clauses can thus be translated to finite set propagators using the proper combination of cartesian product, permutation, projection, inclusion, union, and intersection. Because the cartesian product is the most costly operation, we try to minimize its use and the size of its argument sets.

Chapter 8

Patterns of Interaction and Collaboration

This chapter presents a set of patterns of interacting entities, expressed in SCOLL and analyzed in SCOLLAR. Its intention is to give the reader an idea of the practical applicability of the approach presented in this thesis, not only to analyze patterns in capability systems but also to express, investigate, and compare alternative approaches for building secure software.

We could have provided many more useful and interesting patterns, but we opted for an in depth approach, presenting a complete account of a limited set of patterns, explaining their importance and applicability as well as their SCOLL representation, and analyzing their solutions to the appropriate level of detail.

8.1 Deputies that cannot be Confused

8.1.1 Description of the problem

The problem of confused deputies is very well known by capability proponents. It was first described by Norman Hardy [Har88] and is considered to give proof that permission and designation have to be inextricably combined into unforgeable references: capabilities. Several short explanations of the problem can be found on the Internet [Stia, Mil, Spi, wik].

In this section we will explain the problem from the point of view of applying the principle of least authority (POLA).

The necessary authority to serve a client

When POLA is applied, every entity has no more authority than is necessary to do its job (as was explained in section 1.3.4). When an entity provides services to its clients, the authority it needs will often depend on the client that requests its services. For instance, for a compiler-entity to fulfill a client's request: *"compile-these-files-and-write-the-output-herein"*, it needs read authority to the source files and write authority to the object files.

An upper bound for the service-entity's necessary authority is the union of the authority needed to serve all its possible clients. A practical lower bound for its necessary authority would be: the intersection of the authority needed to serve al its possible clients. In the example of the compiler, this lower bound could be: read/write access to a file that contains site-specific data and hints to optimize the compilation process and output. The upper bound would add to this: read access to all source files and write access to all object files.

To comply with POLA, what authority should we give to the service-entity then? The lower bound is definitely not enough and the upper bound is usually way too much!

Delegation: Just-In-Time Least Authority

If the clients can delegate part of their authority to the service-entity, the problem of least authority can be solved to a first approximation. A service-entity to whom the clients must delegate part of their authority, is called a deputy.

We divide the authority necessary to perform the service into a part that is controlled by the deputy and a part that is controlled by the client, so that neither of them has excess authority. Authority is controlled by a subject, if it depends on the subject's behavior, in other words, if the subject would refuse to use it, it would not be available to other subjects either. The client has to delegate authority to the deputy, in order to enable the deputy to do his job. That means that the client controls part of the authority: he could in principle refuse to delegate. The deputy also controls part of the authority: he could in principle refuse to use his own authority or the delegated authority.

Table 8.1 shows an overview of the authority we consider here.

Table 8.1: Client-controlled versus deputy-controlled authority

authority	controlled by client	controlled by deputy
needed for the service	(1)	(3)
not needed for the service	(2)	(4)

We make sure that (3) contains exactly what the deputy needs for his own accounts, when performing the service. That may even be a strict subset of the practical lower bound we considered earlier. For instance: even when all the compiler's clients would have read authority to a common library of source files, that compiler's least authority (3) would not include read-authority to that library, but only its authority to read and write its proper file with optimization hints.

If all goes well, the client delegates (1) to the deputy, who then uses both (1) and (3) to perform the service. Delegation saved our day: it allows us to give the deputy less than the least authority to do his job, (it can no longer do its job all by itself) and we rely on the clients to provide what more is needed.

But did we not just open Pandora's box, by introducing delegation? How are we going to prevent that everybody starts delegating every authority to everybody else? How will we ever be able to impose confinement of authority in such a chaos?

Although this is not an easy question, there is no reason to panic. In fact, section 8.1.5 will show with a real life example that, since delegation allows us to better comply to POLA, this advantage overshadows the disadvantages. Here we will only discuss this concrete context: the client controls (1) and (2) and can delegate them to the deputy. The deputy controls (3) and (4) but does not want to delegate them. The deputy only wants to make (3) available indirectly to its clients: to be used by the deputy for his own sake, while serving its clients needs.

Confusing the Deputy

The deputy should only use (3) on behalf of the client that delegated (1) to him, and never use (4) on behalf of a client. A deputy is confused if he can be tricked by a client to use his own authority (3) or (4) instead of (1). That can happen, for instance in the compiler example, when the client, instead of designating a proper object file for the output, designates the file that the compiler uses to manage its optimization tables for that purpose. The compiler has indeed the right the write output to that file. The result is a disaster: the compiler was "lured"[1] into overwriting his own optimization file with output of a compilation process. This example is close to the original confused deputy problem that was reported by Norman Hardy [Har88].

The problem is not simple. It has to do with the intention of the deputy: what he wants to use (3) and (4) for. How can we make sure that the deputy uses (3) and (4) only for the right purpose?

8.1.2 Proposed Solutions

The approaches that have been proposed in the literature to solve this problem, can be divided in two categories.

1. Identify the client or his permissions at runtime and make sure that he has permissions that provide him with the authority (1). If necessary, switch off the deputy's permissions that provide him with authority (3) and (4).

2. Make the client's authority (1) portable, so that the deputy can use (1) in the same way the client would, and make passing-portable-authority the only way to delegate. The client then has to explicitly pass the portable authority (1), for the deputy to use. The client will not be able do that unless he has (portable) authority himself. Then rely upon the deputy not to use (3) or (4) instead of (1).

We will now have a brief look into the practical feasibility of both approaches.

Approach 1: Check Who's Asking

If we can check and manipulate the permissions of all subjects at runtime, we can safely proceed as follows: provide the necessary permissions for (1) and (3) to the deputy, only if we can detect that he does not want to use (3) for its client's purposes, as a replacement for (1).

There are theoretical and practical problems with this approach. The most important theoretical problems are:

- What do we mean ... "Who is asking?" ?
  If I delegate a task to you and you delegate it further to the eventual deputy, which one of us is the deputy working for then?

- We cannot infer the deputy's intentions for using (3) from the fact that someone invoked him with or without (1).

Instead of solving these problems exactly, the practical implementations of these approaches settle for the following approximations:

[1]The Confused Deputy attack is sometimes referred to as the "luring attack".

- Assume that the deputy is working for a subset of all potential clients in the call stack and demand that all of the subjects in that subset have permissions that prove their authority (1). For instance, if I call you and you call the deputy, we are both his clients.

- Approximately infer the deputy's intentions for using (3) in some way from the information that you do have. Such approximations are usually crude.

Most practical implementations struggle with either accuracy, safety, administrative overhead, or with a combination of the three.

However, a remarkable approach was taken by Wallach in [WBDF97], in which he proposes the practice of "stack walking". The call stack of the programming language runtime (in casu the Java VM), can be adorned by the caller to indicate if he wants to delegate his own permissions to the called subjects down the call stack or not. The delegation of permissions down the stack can be interrupted or otherwise influenced by the other subjects down the stack, and so on.

The fact that the subjects themselves are given the choice to delegate is an important step towards recognizing that the problem is one of intention and can only be solved with the cooperation of the deputy himself, as only he can know his intentions.

But the approach is only applicable for stack-based implementations. It excludes for instance inter-process and inter-thread calls in a parallel or distributed context. The fact that subjects are allowed to influence information on the call stack opens another can of worms, as the stack itself can now become an extra overt channel for data communication, which conflicts with concerns for data confinement.

In section 8.5.4 we will model stack walking in SCOLL. That will allow us to present the approach formally, and to better explain its advantages and disadvantages.

Approach 2: Capabilities

It is the deputy's responsibility to make sure that its clients delegate their authority (1) to it and to use all authorities (1),(3), and (4) for their proper purpose. All we have to do is : make sure that the deputy has the necessary information to take that responsibility.

Capabilities (Chapter 4) combine designation with access-permission: you can never have one without the other. Access permission to a subject is the permission to use that subject. That means that, like designations, permissions have to be portable. If Alice wants to communicate her reference-to-Carol (designation) to Bob, because she wants Bob to be able to designate Carol, she inevitably delegates her permission-to-use-Carol to Bob at the same time. But it also means that, like permissions, designations have become unforgeable: you can no longer designate any subject by guessing its name (or location). Strings are guessable and will no longer be valid designations.

In capability systems all permissions are packed into capabilities. We no longer need a central structure (e.g. the call stack) to provide the deputy with all the necessary information to check if its clients have authority (1), because the deputy can rely on it that it is impossible for a client to designate a subject without having access-permission to that subject.

Checking a capability is almost as simple as using it, except that now the deputy also has to protect the service it provides to its proper clients from being broken by a rogue (or buggy) client who gives him the wrong kind of capabilities, e.g. carrying authority from (2).

(*This is a concern about defensive consistency that we already encountered in DVH capabilities (section 4.1.6) : to allow data abstractions and services to be used by several clients, while ensuring that these clients remain protected from each other.*)

In the compiler example: the compiler's clients cannot designate the compiler's private optimization file if they don't have permission to use it.

What happens in case the client has capabilities that provide authority (3) or (4), and gives them to the deputy instead of the capabilities for (1). In that case, the deputy is not confused: he uses the capabilities provided by that client just as if they were providing authority (1). In the compiler example: the client that has access to the compiler's optimization file can tell the deputy to overwrite it with compiler output. The client had the full right to use (and abuse) that file anyway, which he probably should not have had. A confinement analysis should be done to find out how that particular client can be prevented from having authority (3) in a portable form.

8.1.3 Capability Based Deputies in SCOLLAR

Let us now investigate what behavior restrictions the deputy should respect in practice, in a capability based system, in order to avoid being "confused". The requirement that the client has to delegate his authority (1) to the deputy by sending a suitable capability may be enough to make confusion avoidable, but that does not necessary mean that avoiding confusion is trivial. SCOLLAR will help us look for the deputy's necessary behavior restrictions.

We will apply two different models for the capability rules: one with simple behavior `may.return` and one with refined `may.returnFor` behavior. The system rules in both models have been explained earlier (Sections 5.5 and 6.5.3).

The initial access graph is shown in figure 8.1. The subject `client` has access to a file `cFile` and to the subject `deputy`. The `deputy` has access to a file `dFile`. Every entity is allowed to have access to itself. The relevant authority is here: what entities in the pattern can the deputy be lured into using as if it was the client's file.

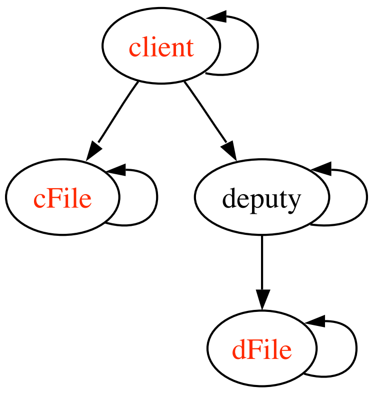


Figure 8.1: The initial access graph for the confused deputy analysis.

To specifically track this authority, we add a rule in the deputy's behavior, `DEPUTY`, that generates private knowledge. We call this knowledge predicate: `useForClient`.

The liveness property states that the deputy should use the client's file for the client's purpose, (in casu, overwrite it with output data). The safety property states that the deputy should not use his own file for that purpose.

Table 8.2 shows the deputy pattern expressed in SCOLL using the simpler variant of the capability interaction model. Some alternatives we will also investigate are indicated between comment brackets (alternatives A and B).

A refined set of predicates and and extended set of system rules for this pattern will investigated also, and is referred to alternative 1 (Table 8.3). In this refined pattern there is no need to add a rule for `may.returnFor` and `may.returnFor0` in the `UNKNOWN` behavior, because the behavior refinement rule takes care of this. There will be no need to add or modify the `deputy`'s behavior either, because the knowledge refinement rules make sure that refined knowledge implies the unrefined knowledge the `deputy` uses to define its behavior.

We first assign `UNKNOWN` behavior to the deputy's `dFile`. That makes the pattern suitable in situations where the deputy did not create that file himself and therefore may not rely on the typically restricted (passive) behavior of that file.

As indicated in the subject section, we will also analyze the alternatives for the behavior of `dFile`:

A) `MINIMUM` behavior: the deputy's file is relied upon to be completely unsuitable for propagating subjects (only data).

B) `MINIMUM` behavior prefixed by the search indicator: `?`
In that variant, SCOLLAR will look for minimal sets of restrictions for the behavior of both the deputy and his "file" subject.

8.1.4 Analysis of the SCOLLAR results

This section represents and discusses the results of the SCOLLAR analysis of the patterns described in section 8.1.3. We used SCOLLAR in its second operation mode (See "Solution Mode" in section 7.2.2). In short, we made SCOLLAR search for all maximal sets of behavior predicates (for the deputy, and in alternative `B` also for its file) to discover what restrictions are necessary and sufficient in the deputy subject's behavior to guarantee that it will not use its own file in the wrong way (as if it was a client-provided file), while not preventing him from using the client-provided file for that purpose.

We will learn two things from the analysis results. First of all, we will learn what behavior we should avoid when implementing or designing a deputy entity that cannot be confused. But there is another important lesson to be learned here: from looking at the deputy's internal knowledge (his intentions), we will find a very important clue about how well we expressed the problem itself in SCOLLAR.

Behavior restrictions from the original set-up

The pattern in table 8.2 returned the access graph in figure 8.2 and a single solution for the deputy's behavior, listed in table 8.5.

From the access graph we can immediately see that nobody got direct access to the `dFile`, even though we did not specify that as a safety property. The other three subjects can all safely be introduced to each other.

The list of behavior restrictions for `deputy` is shown in table 8.4

When we click on the button for the only solution, the behavior and knowledge facts of all the subjects in the pattern are shown in their own table. From that page, table 8.5 shows the extract that contains the deputy's knowledge facts (top part) and

Table 8.2: Deputy Pattern in Scoll

```
declare
    permission:  access/2
    behavior:  may.sendTo/3 may.getFrom/2 may.return/2
               may.receive/1
    knowledge:  did.sendTo/3 did.getFrom/3 did.return/2
                did.receive/2
system
    B:may.receive() A:may.sendTo(B,X) access(A,B)
     access(A,X) => A:did.sendTo(B X);
    A:did.sendTo(B X) => B:did.getFrom(X);
    B:did.getFrom(X) => access(B,X);
    A:may.getFrom(B) B:may.return(X) access(A,B)
     access(B,X) => A:did.getFrom(B,X);
    A:did.getFrom(B,X) => access(A,X);
    A:did.getFrom(B,X) => B:did.return(X);
behavior
    UNKNOWN {
     => may.receive() may.getFrom(A);
     => may.return(X) may.sendTo(A,X);}
    DEPUTY {did.receive(F) => useForClient(F);}
    MINIMAL {}
subject
    client:  UNKNOWN
    cFile:  UNKNOWN
    ?  deputy:  DEPUTY
    dFile:  UNKNOWN
            /* alternative A : dfile:  MINIMAL
            alternative B : ?dfile:  MINIMAL */
config
    access(client,client) access(cFile,cFile)
    access(deputy,deputy) access(dFile,dFile)
    access(client,cFile) access(client,deputy)
    access(deputy,dFile)
goal
    useForClient(deputy,cFile)
     !useForClient(deputy,dFile)
```

Table 8.3: Deputy Pattern Alternative Parts

```
declare
   permission:  access/2
   behavior:  may.sendTo/3 may.getFrom/2 may.return/2
              may.receive/1
              may.returnFor0/2 may.returnFor/3
   knowledge:  did.sendTo/3 did.getFrom/3 did.return/2
               did.receive/2
               did.returnFor0/3 did.returnFor0/2
               did.getFromFor0/3 did.getFromFor/4
system
   B:may.receive() A:may.sendTo(B,X) access(A,B)
    access(A,X) => A:did.sendTo(B X);
   A:did.sendTo(B X) => B:did.getFrom(X);
   B:did.getFrom(X) => access(B,X);
   A:may.getFrom(B) B:may.return(X) access(A,B)
    access(B,X) => A:did.getFrom(B,X);
   A:did.getFrom(B,X) => access(A,X);
   A:did.getFrom(B,X) => B:did.return(X);
   /* ADDITIONAL RULES */
   B:may.returnFor(X,Y) A:may.sendTo(B,X) B:may.receive()
    access(B,Y) access(A,X) access(A,B)
    => A:did.getFromFor(B,X,Y);
   A:did.getFromFor(B,X,Y) => B:did.returnFor(X,Y);
   A:did.getFromFor(B,X,Y) => access(A,Y);
   B:did.returnFor(X,Y) => access(B,X);
   A:may.getFrom(B) B:may.returnFor0(Y) access(A,B)
   access(B,Y)
   => B:did.returnFor0(Y) A:did.getFromFor0(B,Y);
   B:may.return(Y) =>
    => B:may.returnFor(X,Y) B:may.returnFor0(Y);
   A:did.getFromFor0(B,Y) => A:did.getFrom(B,Y);
   B:did.returnFor0(Y) => B:did.return(Y);
behavior
   ...
subject
   ...
config
   ...
goal
   ...
```

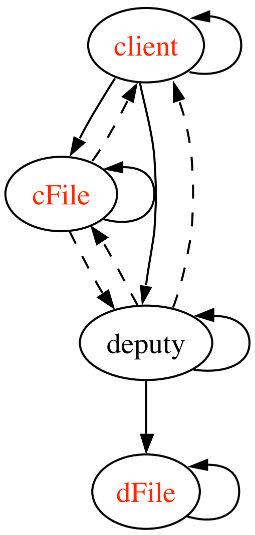


Figure 8.2: The resulting access graph

Table 8.4: The deputy's behavior restrictions.

Solutions	1
deputy:may.sendTo(cFile,dFile)	**0**
deputy:may.sendTo(client,dFile)	**0**
deputy:may.sendTo(dFile,cFile)	**0**
deputy:may.sendTo(dFile,client)	**0**
deputy:may.sendTo(dFile,deputy)	**0**
deputy:may.sendTo(deputy,dFile)	**0**
deputy:may.return(dFile)	**0**

behavior facts (bottom part) in matrix form. The columns in that matrix indicate the subject in the last argument of the fact.

Table 8.5: The deputy's knowledge and behavior (1)

deputy				
	client	cFile	deputy	dFile
access(deputy,_)	1	1	1	1
deputy:did.receive(_)	1	1	1	_
deputy:did.return(_)	_	_	_	_
deputy:did.getFrom(client,_)	1	1	_	_
deputy:did.getFrom(cFile,_)	1	1	_	_
deputy:did.getFrom(deputy,_)	1	1	_	_
deputy:did.getFrom(dFile,_)	_	_	_	_
deputy:did.sendTo(client,_)	_	_	_	0
deputy:did.sendTo(cFile,_)	1	_	1	0
deputy:did.sendTo(deputy,_)	1	1	1	0
deputy:did.sendTo(dFile,_)	0	0	0	_
deputy:useForClient(_)	1	🟩1	1	🟥0
deputy:may.receive()	1			
deputy:may.getFrom(_)	1	1	1	1
deputy:may.return(_)	1	1	1	**0**
deputy:may.sendTo(client,_)	_	_	_	**0**
deputy:may.sendTo(cFile,_)	1	_	1	**0**
deputy:may.sendTo(deputy,_)	1	1	1	**0**
deputy:may.sendTo(dFile,_)	**0**	**0**	**0**	_

The goal properties (safety and liveness) are part of the deputy's knowledge (upper part of the table). The safety property is indicated with 🟥0, the liveness property with 🟩1. They indicate that the solution is indeed safe and alive.

The behavior predicates marked with **0** indicate what the deputy should not do. Let us consider them one by one:

`deputy:may.return(dFile)` The deputy should not return his own file to his clients as part of his service. A restriction that can indeed be expected.

`deputy:may.sendTo(client,dFile)` Of course, the deputy should not simply send his own file to the client. Otherwise the client could afterwards designate that file for the deputy to overwrite. The reader is correct to ask himself : "How did the deputy got access to the client anyway?"

The client has access to himself, and could have provided himself to the deputy instead of his file. That is completely legal and realistic, as the client may implement a file interface himself and cope with the overwrite intentions of the deputy himself.

This suggest a nice way for a client to test what a deputy would do to his real file: provide himself (or one of his allies) to the service and see what happens. Of course such a test would not be fail safe: the deputy may not be that predictable.

`deputy:may.sendTo(cFile,dFile)` The deputy should not send his own file to the client's file subject. This restriction holds a useful warning about the client's file: since we don't wanted to rely on its behavior, SCOLLAR assumed (as we should too) that it can as well be an adversary, cleverly disguised as (with the interface of) a file.

`deputy:may.sendTo(deputy,dFile)` This restriction needs some explanation. At first sight it may seem to make no sense: why should the deputy not be allowed to send his own file to himself? Because then he may confuse himself: as a responder he may assume that a client did send that file to him in which case he will overwrite that file.

If our deputy really needs to send that file to himself, then he should be able to make the difference. His behavior in our pattern must be adapted to reflect how he makes that difference. At the current level of detail, SCOLLAR warns for the imminent danger of the deputy confusing himself!

`deputy:may.sendTo(dFile,client)` This says: the deputy should not send the client to dFile. This makes sense if `dFile` is not a simple file, which is a realistic assumption. Instead of a simple file, many deputies want to use a deputy themselves. The restriction points out the danger that this second-level deputy (`dFile`) may not be restricted enough. For instance, if such a `dFile` gets access to `client`, `dFile` may become known to `client` who can then "confuse" the deputy.

`deputy:may.sendTo(dFile,cFile)` The deputy should not send the client's file to `dFile`. The reason is the same as in the previous restriction: `dFile` may become known to `client`.

`deputy:may.sendTo(dFile,deputy)` The deputy should not send himself to `dFile`, because `dFile` itself could then play client, and make the deputy overwrite himself. If this would be an intended attack by `dFile`, the deputy's behavior restriction would be futile, because `dFile` could as well overwrite himself. But in case "unknown behavior" really means what it says (and not necessary "hostile behavior"), the restriction certainly makes sense.

Other things we can learn from the analysis

It is often useful to look at the resulting tables, both at the knowledge part and at the behavior part, to learn interesting things about the pattern itself as well as about how the pattern was expressed in SCOLLAR.

An obviously useful view is presented by the graph of access relations between the subjects in the pattern. This can be generated in SCOLLAR via its interface to the graph visualization tool GraphViz [KN93, GN00, GV05]. For the current example, figure 8.2 showed the resulting graph.

But to understand a pattern and the role an entity plays in it, it can be useful to look a the individual fields in that subject's fact table. Even in this seemingly simple pattern, there are things to discover.

Look for instance at the deputy's internal knowledge facts `useForClient`. We already discussed two of them in detail: `deputy:useForClient(cFile)` is what we wanted to be possible (liveness) and `deputy:useForClient(dFile)` is what we regarded as confusion and wanted to be impossible.

A third one, `deputy:useForClient(client)` turns our attention again to the possibility that the client presents itself as his own file. We saw that there was nothing wrong with that. The interested reader is invited to try if that could be prevented, by adding this predicate as a safety property to the *goal* part of the pattern.

We want to draw the attention of the reader to the remaining fact of this predicate: `deputy:useForClient(deputy).` It literally says there: the deputy will treat itself as if it was a file to be used on the clients behalf. That was not our intention, or was it? Of course, the client can indeed provide the deputy to the deputy, because he has access to the deputy. That means that the deputy can always be “lured” into treating himself as a file delegated by its client!

The deputy will have to counter his own intentions to use itself as a client’s file. The easiest way to do that is to make sure that trying to write to it has no effect. After all, we are working in a collaborative model and the deputy as an invoker of his own “write output” routine has no authority to actually write anything, unless the deputy as a responder (implementor of that routine) wants to collaborate. The simplest thing is to make sure that the deputy does not implement such a routine or method, by encapsulating the deputy’s “write output” call in an error handling routine and by catching the error that will be thrown.

The interested reader is encouraged to model this refined behavior in Scoll, using for instance the extension for data propagation discussed in section 5.6.3.

Alternative 1: using refined behavior

The extended pattern using the rules of table 8.3 also returned a single solution for the deputy’s behavior. The access graph is the same as before (figure 8.2).

The list of restrictions to be imposed on the deputy’s behavior is presented in table 8.6. It contains the same restrictions as before, plus restrictions for the refined predicates of `may.return`.

Table 8.6: The deputy’s behavior restrictions when using refined behavior. (Alt. 1)

Solutions	1
deputy:may.sendTo(cFile,dFile)	**0**
deputy:may.sendTo(client,dFile)	**0**
deputy:may.sendTo(dFile,cFile)	**0**
deputy:may.sendTo(dFile,client)	**0**
deputy:may.sendTo(dFile,deputy)	**0**
deputy:may.sendTo(deputy,dFile)	**0**
deputy:may.return(dFile)	**0**
deputy:may.returnFor0(dFile)	**0**
deputy:may.returnFor(cFile,dFile)	**0**
deputy:may.returnFor(client,dFile)	**0**
deputy:may.returnFor(deputy,dFile)	**0**

Table 8.6 shows that of the five refinements of `deputy:may.return(dFile)` only four are restricted: `deputy:may.returnFor(dFile,dFile)` is not in the list. This means that it is OK for the `deputy` to return `dFile` in exchange for `dFile` in the same invocation. However, because `deputy` is the only one who has access

to `dFile`, and `deputy:may.sendTo(deputy,dFile)` is forbidden, such an exchange will never take place. The behavior is irrelevant in this pattern.

The zero's in the lower three rows of table 8.6 simply enforce the earlier restriction in `deputy:may.return(dFile)` that the deputy should never return his own file, not even in exchange for anything else.

Alternative A: relying on the deputy's file

We now take the original pattern in table 8.2 and set `dFile`'s behavior to `MINIMUM` to express that we can rely on the fact that `dfile` is a simple file. Now SCOLLAR also finds a single solution for the deputy's behavior, its access graph being the same as before (figure 8.2).

The restrictions to be imposed on the deputy's behavior are listed in table 8.7.

Table 8.7: The deputy's behavior restrictions when relying upon `dFile`. (Alt. A)

Solutions	①
deputy:may.sendTo(cFile,dFile)	**0**
deputy:may.sendTo(client,dFile)	**0**
deputy:may.sendTo(deputy,dFile)	**0**
deputy:may.return(dFile)	**0**

Compared to the results in table 8.4, where `dFile` had `UNKNOWN` behavior, no restrictions were added (as could be expected), and out of the seven restrictions listed there, the following four were kept (for the reader's convenience, the comments are repeated here):

`deputy:may.return(dFile)` A restriction that can indeed be expected: the deputy should not return his own file to his clients as part of his service.

`deputy:may.sendTo(cFile,dFile)` The deputy should not send his own file to the client's file. This restriction holds a useful warning about the client's file: we don't know anything about it's behavior. It could be a simple file, but it could also be an adversary, cleverly disguised as a file.

`deputy:may.sendTo(client,dFile)` Of course, the deputy should not simply send his own file to the client. Otherwise the client could afterwards designate that file for the deputy to overwrite. The reader is correct to ask himself : "How did the deputy got access to the client anyway?"

The client has access to himself (by reflexive access rule), and could have provided himself to the deputy instead of his file. That is completely legal and realistic, as the client may implement a file interface himself and cope with the overwrite intentions of the deputy himself. It suggest a nice way for a client to test what a deputy would do to his real file: provide himself self (or one of his allies) to the service and see what happens. Of course such a test would not be fail safe: the deputy may not be that predictable.

`deputy:may.sendTo(deputy,dFile)` This restriction needs some explanation. At first sight it may seem to make no sense: why should the deputy not be allowed to send his own file to himself? Because then he may confuse himself: as a responder, he may assume that a client did send that file to him and overwrite it.

If our deputy really needs to send that file to himself, then he should be able to make the difference, and then his behavior in our pattern must be adapted to reflect how he makes that difference. At the current level of detail, SCOLLAR warns for the imminent danger of the deputy confusing himself!

Compared to the results in table 8.5, where `dFile` had `UNKNOWN` behavior, the following restrictions are no longer relevant:

```
deputy:may.sendTo(dFile,cFile)

deputy:may.sendTo(dFile,client)

deputy:may.sendTo(dFile,deputy)
```

Since `dFile` no longer cooperates in any way, it is no longer relevant what the deputy's behavior wants to send to it.

Alternative 1A: relying on the deputy's file and using exchange rules

For completeness, we briefly mention the results of combining alternative `1` with alternative `A)`. Table 8.8 shows the results. Again, a single solution was found by SCOLLAR and the access graph does not differ from the previous ones.

Table 8.8: The deputy's behavior restrictions with refined behavior and restricted `dFile` (Alt. 1A)

Solutions	①
deputy:may.sendTo(cFile,dFile)	**0**
deputy:may.sendTo(client,dFile)	**0**
deputy:may.sendTo(deputy,dFile)	**0**
deputy:may.return(dFile)	**0**
deputy:may.returnFor0(dFile)	**0**
deputy:may.returnFor(cFile,dFile)	**0**
deputy:may.returnFor(client,dFile)	**0**
deputy:may.returnFor(deputy,dFile)	**0**

The reader is invited to check that this list adds the restrictions to alternative `A` that were also added to the original set up by alternative `1`.

Alternative B : maximizing `dFile`'s behavior too.

We now instruct SCOLLAR to search for minimal combinations of restrictions in the behavior of both the deputy and his file. Figure 8.3 shows the combined access graph for all the solutions.

In the graph we see three extra access arcs (the dotted ones) that were not possible before, when `dfile` was either completely unknown or a simple file. This shows the importance of well balanced behavior restrictions: relying upon a collaborator's well

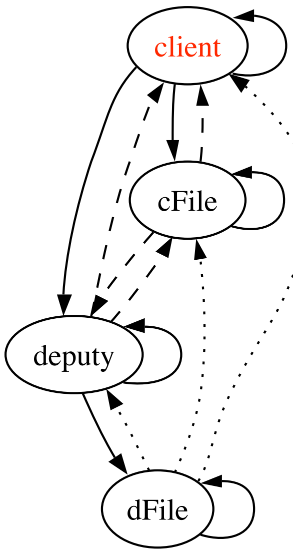


Figure 8.3: The combined access graph of all the solutions.

balanced behavior opens up possibilities for safe collaboration that would be impossible using a completely restricted or completely unrestricted collaborator.

It also shows the importance of modeling behavior approximations precisely. Coarse approximations, like “all or nothing”, lead to less solutions.

SCOLLAR finds 12 solutions, imposing restrictions on 15 behavior facts, as is presented in table 8.9. The table is split up in restrictions for `dFile` (top part) and restrictions for `deputy` (bottom part).

The deputy’s restrictions in solutions 1 and 12 correspond to table 8.7, where `dFile`’s behavior was `MINIMAL`. This means that we can relax the strict passiveness of `dFile`, either as is proposed by solutions 1 or by solution 12, without necessarily having to restrict the deputy’s behavior any further. For that it suffices that the `dFile` either does not send himself (solution 1), or does not receive anything as a responder (solution 12).

Solution 2 represent the situation in which `dFile`’s behavior is unrestricted. The reader can verify that solution 2 corresponds to the unique solution in table 8.4, where `dFile`’s behavior was `UNKNOWN`.

Notice that 11 out of the 12 solutions restrict `deputy` in a different way. This means that relying on `dFile` gives us 11 times as many possibilities for the `deputy`’s maximal behavior than we had if we did not rely on `dFile`.

Note also that the deputy’s behavior restrictions found in table 8.7, where `dFile`’s behavior was `MINIMAL`, are consistently required in all solutions. This can be easily checked by looking at the four rows in table 8.9 that contain all zero’s.

There is exactly one solution in which the access graph actually includes all dotted arcs in figure 8.3. The reader is encouraged to try this example in SCOLLAR’s online version and find the unique solution that puts `dFile` in the unique position of having access to all entities in the pattern.

Table 8.9: The 12 solutions restricting the behavior of `deputy` and `dFile`. (Alt. B)

Solutions	1	2	3	4	5	6	7	8	9	10	11	12
dFile:may.getFrom(cFile)	1	1	1	1	**0**	**0**	**0**	1	1	1	**0**	1
dFile:may.getFrom(client)	1	1	1	1	1	1	1	**0**	**0**	**0**	**0**	1
dFile:may.getFrom(deputy)	1	1	1	**0**	1	1	**0**	1	1	**0**	1	1
dFile:may.sendTo(cFile,dFile)	**0**	1	1	1	**0**	**0**	**0**	1	1	1	**0**	1
dFile:may.sendTo(client,dFile)	**0**	1	1	1	1	1	1	**0**	**0**	**0**	**0**	1
dFile:may.sendTo(deputy,dFile)	**0**	1	**0**	**0**	1	**0**	**0**	**0**	1	**0**	1	1
dFile:may.receive()	1	1	1	1	1	1	1	1	1	1	1	**0**
deputy:may.sendTo(cFile,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.sendTo(client,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.sendTo(dFile,cFile)	1	**0**	**0**	**0**	1	1	1	**0**	**0**	**0**	1	1
deputy:may.sendTo(dFile,client)	1	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1
deputy:may.sendTo(dFile,deputy)	1	**0**	1	1	**0**	1	1	1	**0**	1	**0**	1
deputy:may.sendTo(deputy,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.return(cFile)	1	1	**0**	1	1	1	1	**0**	1	1	1	1
deputy:may.return(client)	1	1	**0**	1	1	**0**	1	1	1	1	1	1
deputy:may.return(dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Alternative 1B : optimizing `dFile`'s behavior with *exchange*.

We now consider the refined predicates and rules and let SCOLLAR search for minimal combinations of restrictions in the behavior of both the deputy and his file.

The combined access graph for all solutions is depicted in figure 8.4 and is not different from the one in figure 8.3. The behavior refinement did not influence the maximal propagation of access permissions.

SCOLLAR finds 25 solutions before timing out after 30 seconds, imposing restrictions on 34 behavior facts, that are presented in tables 8.10 and 8.11. The tables are split up in restrictions for `dFile` (top part) and restrictions for `deputy` (bottom part).

There is again exactly one solution in which the access graph actually includes all dotted arcs in figure 8.4. The reader is encouraged to try this example in SCOLLAR's online version and find the unique solution that gives `dFile` access to all entities in the pattern.

Again we can make the comparison with the situation in alternative 1A (table 8.8), where refined behavior predicates and rules were used and where `dFile`'s behavior was `MINIMUM`. All seven restrictions in the deputy's behavior can be consistently found in all solutions of tables 8.10 and 8.11.

8.1.5 "Little Snitch" : A User Experience with Reference Monitors

The Mac OS X program Little Snitch ®[lit] provides a pedagogical example that allows you to watch reference monitoring at work, and to detect the need for unconfusable deputies. It allows the user to grant (and revoke) a program *this-time*, *this-session*, or *permanent* access to the Internet. The *snitch* is configured by default to ask the user if he wants to give a program access to the Internet, the first time the that program tries to connect to the Internet. Programs the user denies access to the Internet to are made

Table 8.10: Solutions 1 to 13. (Alt. 1B)

Solutions	1	2	3	4	5	6	7	8	9	10	11	12	13
dFile:may.getFrom(cFile)	1	1	1	**0**	**0**	1	1	1	1	1	1	1	1
dFile:may.getFrom(client)	1	1	1	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
dFile:may.getFrom(deputy)	1	1	**0**	1	**0**	1	1	1	1	1	**0**	**0**	**0**
dFile:may.sendTo(cFile,cFile)	1	_	_	**0**	**0**	1	1	1	1	1	1	1	1
dFile:may.sendTo(cFile,client)	1	_	_	_	_	_	_	_	_	_	_	_	_
dFile:may.sendTo(cFile,dFile)	**0**	1	1	**0**	**0**	1	1	1	1	1	1	1	1
dFile:may.sendTo(cFile,deputy)	1	_	_	_	**0**	1	1	1	1	1	1	1	1
dFile:may.sendTo(client,cFile)	1	_	_	_	_	_	_	_	_	_	_	_	_
dFile:may.sendTo(client,client)	1	_	_	_	_	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
dFile:may.sendTo(client,dFile)	**0**	1	1	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
dFile:may.sendTo(client,deputy)	1	_	_	_	_	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**
dFile:may.sendTo(deputy,cFile)	1	_	_	_	**0**	1	1	1	1	1	1	1	1
dFile:may.sendTo(deputy,client)	1	_	_	_	_	**0**	1	**0**	1	1	**0**	**0**	1
dFile:may.sendTo(deputy,dFile)	**0**	1	**0**	1	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**
dFile:may.sendTo(deputy,deputy)	1	_	**0**	1	**0**	1	1	**0**	**0**	1	**0**	1	1
deputy:may.sendTo(cFile,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.sendTo(client,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.sendTo(dFile,cFile)	1	**0**	**0**	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.sendTo(dFile,client)	1	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1
deputy:may.sendTo(dFile,deputy)	1	**0**	1	**0**	1	1	1	1	1	**0**	1	1	1
deputy:may.sendTo(deputy,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.return(cFile)	1	1	1	1	1	**0**	**0**	**0**	**0**	1	1	**0**	**0**
deputy:may.return(client)	1	1	1	1	1	1	1	1	1	1	1	1	1
deputy:may.return(dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.returnFor0(cFile)	1	1	1	1	1	**0**	**0**	**0**	**0**	1	1	1	1
deputy:may.returnFor0(client)	1	1	1	1	1	1	1	1	1	1	1	1	1
deputy:may.returnFor0(dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.returnFor(cFile,client)	1	1	1	1	1	1	1	1	1	1	1	1	1
deputy:may.returnFor(cFile,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.returnFor(client,cFile)	1	1	1	1	1	1	**0**	1	**0**	1	1	1	**0**
deputy:may.returnFor(client,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.returnFor(deputy,cFile)	1	1	1	1	1	**0**	**0**	1	1	1	1	**0**	**0**
deputy:may.returnFor(deputy,client)	1	1	1	1	1	1	1	1	1	1	1	1	1
deputy:may.returnFor(deputy,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Table 8.11: Solutions 14 to 25. (Alt. 1B)

Solutions	14	15	16	17	18	19	20	21	22	23	24	25
dFile:may.getFrom(cFile)	1	**0**	1	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
dFile:may.getFrom(client)	**0**	**0**	1	1	1	1	1	1	1	1	1	1
dFile:may.getFrom(deputy)	**0**	1	1	1	**0**	1	1	1	1	**0**	**0**	**0**
dFile:may.sendTo(cFile,cFile)	1	**0**	1	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
dFile:may.sendTo(cFile,client)	_	**0**	1	1	1	1	1	1	1	1	1	1
dFile:may.sendTo(cFile,dFile)	1	**0**	1	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
dFile:may.sendTo(cFile,deputy)	1	1	1	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
dFile:may.sendTo(client,cFile)	_	**0**	1	1	1	1	1	1	1	1	1	1
dFile:may.sendTo(client,client)	**0**	**0**	1	1	1	1	1	1	1	1	1	1
dFile:may.sendTo(client,dFile)	**0**	**0**	1	1	1	1	1	1	1	1	1	1
dFile:may.sendTo(client,deputy)	**0**	1	1	1	1	1	1	1	1	1	1	1
dFile:may.sendTo(deputy,cFile)	1	1	1	1	1	1	**0**	**0**	1	**0**	1	1
dFile:may.sendTo(deputy,client)	1	1	1	1	1	1	1	1	1	1	1	1
dFile:may.sendTo(deputy,dFile)	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
dFile:may.sendTo(deputy,deputy)	**0**	1	**0**	1	1	**0**	**0**	1	1	1	**0**	1
deputy:may.sendTo(cFile,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.sendTo(client,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.sendTo(dFile,cFile)	**0**	1	**0**	**0**	**0**	1	1	1	1	1	1	1
deputy:may.sendTo(dFile,client)	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.sendTo(dFile,deputy)	1	**0**	1	1	1	1	1	1	1	1	1	1
deputy:may.sendTo(deputy,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.return(cFile)	**0**	1	**0**	**0**	**0**	1	1	1	1	1	1	1
deputy:may.return(client)	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.return(dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.returnFor0(cFile)	1	1	**0**	**0**	1	1	1	1	1	1	1	1
deputy:may.returnFor0(client)	1	1	**0**	**0**	1	**0**	**0**	**0**	**0**	1	1	1
deputy:may.returnFor0(dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.returnFor(cFile,client)	1	1	1	1	1	**0**	1	1	**0**	1	**0**	**0**
deputy:may.returnFor(cFile,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.returnFor(client,cFile)	**0**	1	1	1	1	1	1	1	1	1	1	1
deputy:may.returnFor(client,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
deputy:may.returnFor(deputy,cFile)	1	1	1	**0**	**0**	1	1	1	1	1	1	1
deputy:may.returnFor(deputy,client)	1	1	1	**0**	**0**	1	1	**0**	**0**	**0**	1	**0**
deputy:may.returnFor(deputy,dFile)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

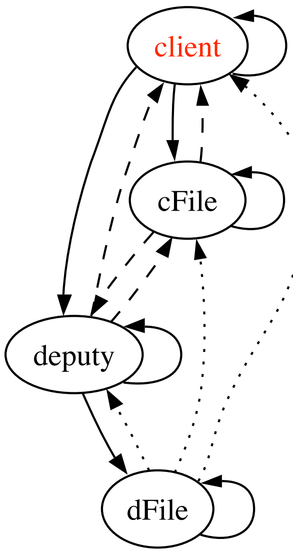


Figure 8.4: The combined access graph of all the solutions.

believe that the computer is not connected to the Internet for the moment.

It is a useful little tool that shows what programs actually make use of the Internet. You would be surprised to see how many programs actually do that, for a reason you as a user can only guess. Is the program calling home to check if your license was expired or pirated? Is the program (infected by) spyware?

It dawned to me that this was an example of how futile reference monitoring can be, when *Little Snitch* asked me whether I wanted to give my *ssh* tool access to the Internet. What should I do? If I only gave it this-session allowance, I would have to repeat that every time. But if I didn't, other programs I did not want to have access could simply call upon *ssh* to make their home-calls!

At first sight, it may seem that this is just another instance of the usability–security trade-off, most users get accustomed to very quickly when being confronted with security. The security part usually gets the worst deal in this situation. Indeed, eventually I gave *ssh* permanent access to the Internet. I am now vulnerable to all spyware that is smart enough to use *ssh* for their Internet access.

Suppose I was more of a paranoiac nature, as a good security researcher should probably be, and I decided to give *ssh* only *this-time* or *this-session* access to the Internet. How will I decide when to give it access and when not to? Little Snitch could possibly be extended to try to find out who called *ssh* and provide me with a complete call stack to help me decide, following the "*check who's asking*"-strategy explained in section 8.1.2. The question then still remains: which one of the programs in the call stack actually wants to use the Internet and for what purpose? Even if I would have the patience I would often not be able to make a well founded decision and I would eventually have to guess.

But suppose now that access-to-the-Internet was a capability and *ssh* was written as a non-confusable deputy that expected its clients (e.g. the user shell) to pass it the capability-to-access-the-Internet as an argument. I could then simply instruct my shell program to pass that capability to the programs I trust, who could use it directly to connect to the Internet, or pass it on to *ssh* if they wanted to.

I would no longer have to make the choice: give *ssh* access to the Internet or not: it would be able to access the Internet exactly when my trusted programs would want it to be.

8.2 Revokable authority

We have seen in Chapter 4 that, in capability based systems, access is an irrevocable permission that comes with every capability. The access permission allows the owner to use the entity designated by the capability. Only the authority that is always guaranteed by the designated entity, is irrevocable authority. For instance, a file-capability will guarantee you the authority to read and write from that file and a read-stream capability will guarantee you the authority to read a certain file.

A read-stream capability can be implemented as an intermediate entity that has access to a file and is programmed to never write to the file or pass on its file capability to other entities but only to use its file capability for reading. Because of its restricted behavior, it reduces the authority it provides to its clients to read-only authority on the file.

In the same way, we can make a revocation assistant that acts as a proxy to a certain capability and stops collaborating when we tell it to. An irrevocable permission to use this assistant provides revocable authority to the actual capability. In this section we will show some SCOLL patterns in which the collaborative behavior of a relied-upon entity is diminished at runtime to revoke authority.

SCOLL does not provide a direct way to express diminished behavior, because all behavior rules as well as all system rules are monotonic. The more knowledge, the more behavior is generated. The behavior of a revocation assistant is non-monotonic: when receiving certain knowledge (revocation is requested) it becomes **less** cooperative.

To circumvent this problem, we examine the maximally reachable propagation of authority before the revocation request to check if revocation is still effective. Revocation is only effective as long as the revocation assistant remains the only entity that can provide authority to use the protected capability.

8.2.1 The Caretaker Pattern

The pattern we investigate here is an adaptation of the one proposed by Redell [Red74], called “the caretaker”. Miller [Mil06b] uses a slightly simplified version of that pattern to explain how capability systems can provide revocable authority by making use of access abstractions, similar to the read-stream entity and the revocation assistant mentioned above. He also shows that analysis based on permissions alone, is too crude to detect the revocability in such patterns, because the crucial influence of behavior is not taken into account.

Here, we investigate a practical aspect of the applicability of the caretaker pattern: what are the conditions we have to rely on to make the revocation in the pattern effective?

Figure 8.5 shows the initial access graph of a caretaker pattern. Subject `alice` wants to give `bob` revokable access to `carol` and has created the `caretaker` for that purpose. The `caretaker`’s behavior is that of a simple proxy to `carol`: passing

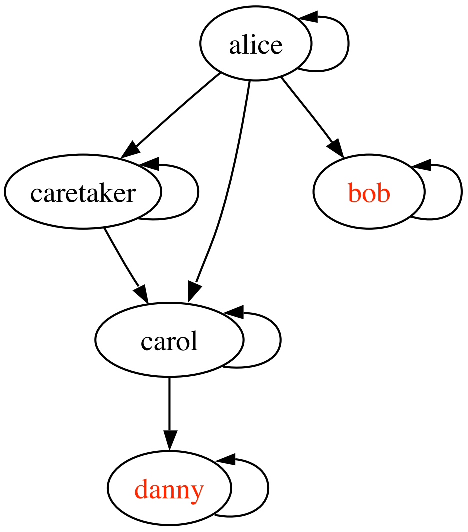


Figure 8.5: The Caretaker Pattern - Initial Configuration

on to `carol` every method and argument and returning to its invoker everything that `carol` returned from the invocation.

An actual implementation of this proxying behavior for the caretaker is given in E-language code [MSC$^+$01] in Miller's [Mil06b]. However, only the code for the caretaker itself is presented there, to show that revocation can work. Programming languages, even capability secure ones like E, are not the most suitable instruments to investigate what behavior restrictions are necessary in a pattern of collaborating entities. To that purpose we will express the pattern in SCOLL and use SCOLLAR to analyze it.

The `caretaker` may always be able to stop collaborating, but that does not necessary revoke `bob`'s authority to use `carol` (remember that authority can flow via many ways). if `alice` and `carol` cannot be relied upon, switching off the `caretaker`'s behavior can be futile. For instance, an unrestricted `alice` may (inadvertently) send or return `carol` to `bob` and thereby give `bob` irrevocable authority to use `carol`. Or `carol`, when programmed wrongly, may return herself and then the `caretaker` will return `carol` to his invoker, again giving `bob` the irrevocable authority to use `carol`.

We added a fifth subject to the pattern: `danny`, an unknown subject, initially accessible by `carol` only. The safety requirement is simple: make sure that `bob` will never get direct access (and therefore irrevocable authority) to `alice`. The liveness property will state that `bob` must be allowed to acquire access to `danny` to make sure that `carol`'s behavior restrictions do not hinder here functionality as a service providing entity.

Table 8.12 shows the caretaker pattern expressed in SCOLL.

Notice in particular how we have expressed the `caretaker`'s proxying behavior exactly as described in section 5.5.5, where we modeled an entity that forwards to an access tester. We want to rely on the `caretaker` to proxy between `carol` just as if it its clients would invoke `carol` directly, even if `carol`'s decision to return a value to her invoker depends on what she received in the same invocation.

Table 8.12: The caretaker pattern in SCOLL

```
declare
  permission:  access/2
  behavior:  may.sendTo/3 may.getFrom/2 may.return/2
             may.receive/1 may.returnFor/3 may.returnFor0/3
  knowledge:  did.sendTo/3 did.getFrom/3 did.return/2
             did.receive/2 did.returnFor/3 did.returnFor0/2
             did.getFromFor/4 did.getFromFor0/3
system
  B:may.receive() A:may.sendTo(B,X) access(A,B) access(A,X)
   => A:did.sendTo(B,X);
  A:did.sendTo(B X) => B:did.getFrom(X);
  B:did.getFrom(X) => access(B,X);
  A:may.getFrom(B) B:may.return(X) access(A,B) access(B,X)
   => A:did.getFrom(B,X);
  A:did.getFrom(B,X) => access(A,X);
  A:did.getFrom(B,X) => B:did.return(X);
  B:may.returnFor(X,Y) A:may.sendTo(B,X) B:may.receive()
   access(B,Y) access(A,X) access(A,B)
   => A:did.getFromFor(B,X,Y);
  A:did.getFromFor(B,X,Y) => B:did.returnFor(X,Y);
  A:did.getFromFor(B,X,Y) => access(A,Y);
  B:did.returnFor(X,Y) => access(B,X);
  A:may.getFrom(B) B:may.returnFor0(Y) access(A,B) access(B,Y)
  => B:did.returnFor0(Y) A:did.getFromFor0(B,Y);
  B:may.return(Y) => B:may.returnFor(X,Y) B:may.returnFor0(Y);
  A:did.getFromFor0(B,Y) => A:did.getFrom(B,Y);
  B:did.returnFor0(Y) => B:did.return(Y);
behavior
  UNKNOWN {=> may.receive() may.getFrom(A);
           => may.return(X) may.sendTo(A,X);}
  PROXY { => may.receive();
         isProxy(P) => may.getFrom(P);
         isProxy(P) did.getFrom(X) => may.sendTo(P,X);
         did.getFromFor0(P,X) => may.returnFor0(X);
         did.getFromFor(P,X,Y) => may.returnFor(X,Y);}
  ALICEMINIMAL { => may.receive();
              isBob(B) isCaretaker(C) => may.sendTo(B,C);}
  MINIMAL {}
subject
  alice:  ALICEMINIMAL bob:  UNKNOWN
  ?  carol:  MINIMAL danny:  UNKNOWN caretaker:  PROXY
config
  access(alice,alice) access(alice,bob) access(alice,carol)
  access(alice,caretaker) isBob(alice bob)
  isCaretaker(alice caretaker) isProxy(caretaker carol)
  access(bob,bob) access(caretaker,caretaker)
  access(caretaker,carol) access(carol,carol)
  access(carol,danny) access(danny,danny)
goal
  access(bob,danny) !access(bob,carol)
```

To let the `caretaker` know who he should proxy to, we initialize his knowledge with the private knowledge fact `caretaker:isProxy(carol)`. In the same way, we give `alice` initial knowledge with
`alice:isBob(bob)` and
`alice:isCareTaker(caretaker)`
because she has to introduce `caretaker` to `bob`.

8.2.2 Maximizing `carol`'s behavior.

Since we want to know what behavior we can allow `carol`, her name is marked with a "?" in the `subject` part of the program.

Figure 8.6 shows the access graph for the combined results. The arcs in solid style represent access in the initial graph. The arcs in dashed style are reachable in all results, the dotted arcs represent access that is reachable in some solutions, but not in others.

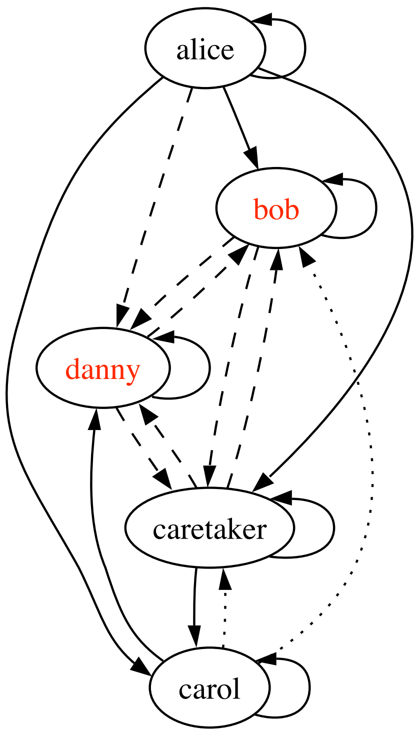


Figure 8.6: The combined access graph

Table 8.13 shows an overview of the results.

Let us check what is interesting about these results, starting with the three absolute restrictions, common to all solutions:

`carol:may.sendTo(danny,carol)` If `carol` would send herself to `danny`, `danny` would be able to pass her on to `bob`.

`carol:may.return(carol)` We suspected this already: `carol` should not return herself, at least not without requiring appropriate "proof of access" to other confined subjects from its invokers.

Table 8.13: The three solutions: possibilities for `carol`'s behavior restrictions

Solutions	①	②	③
carol:may.getFrom(danny)	1	**0**	1
carol:may.sendTo(bob,carol)	**0**	1	**0**
carol:may.sendTo(danny,carol)	**0**	**0**	**0**
carol:may.sendTo(danny,danny)	1	**0**	1
carol:may.receive()	1	**0**	**0**
carol:may.return(carol)	**0**	**0**	**0**
carol:may.returnFor0(carol)	**0**	**0**	**0**
carol:may.returnFor(bob,carol)	**0**	1	1
carol:may.returnFor(caretaker,carol)	**0**	1	1
carol:may.returnFor(danny,carol)	**0**	1	1

`carol:may.returnFor0(carol)` `carol` should not return herself when no "proof of access" to any subject is provided by its invoker in the same invocation.

Let us now check what is specific in the three solutions:

solution 1 : This is the only solution in which `carol` is allowed to accept what is sent to her : `carol:may.receive()`. Therefore this is the only solution that will be useful, if `carol` is to play a general role in the pattern.

It is not surprising that `carol` is not allowed to send herself to `bob` as the effect of such behavior would directly violate the safety property. The other restrictions in this solution are all refinements of `carol:may.return(carol)`.

This solution makes all access depicted in the access graph (Figure 8.6) reachable.

solutions 2 and 3 : These solutions are less uninteresting.

Carol is allowed to return herself to subjects that prove to her that they have access to any subject, by sending her that subject:
`carol:may.returnFor(_,carol)`.

But from the fact that `carol:may.receive()` is forbidden, we can infer that this behavior can never have an effect. When clicking on either of these solutions, the results confirm our suspicion. For the same reason, `carol` cannot get access to anybody but herself, `danny` and the subjects `danny` returns to here.

In solution 2 `carol` is not allowed to request return values from `danny`. Therefore she will not get access to `bob` and `carol:may.sendTo(bob,carol)` is allowed .

8.2.3 Maximizing both `alice`'s and `carol`'s behavior

Let us now find the minimal restrictions for both `alice` and `carol`. Like in alternative `B` of section 8.1.4, the combinations of these restrictions may provide more interesting solutions.

We first make the following adaptations to the SCOLL pattern of table 8.12:

1. Add a “?” mark before `alice`’s name in the `subject` part.
2. To make sure that `alice` and `carol` have sensible minimum behavior, add the following rules to both their behavior:

   `=> may.receive()` (*alice’s behavior already had this rule*)

   `did.receive(X) => may.returnFor(X,X)`
3. To make sure that `bob` never gets access to `danny` directly from `alice` or `carol`, add the following safety properties to the SCOLL pattern:

   ```
   !  alice:did.sendTo(bob,danny)
   !  carol:did.sendTo(bob,danny)
   !  bob:did.getFrom(alice,danny)
   !  bob:did.getFrom(carol,danny)
   ```

The resulting graph is shown in figure 8.7.

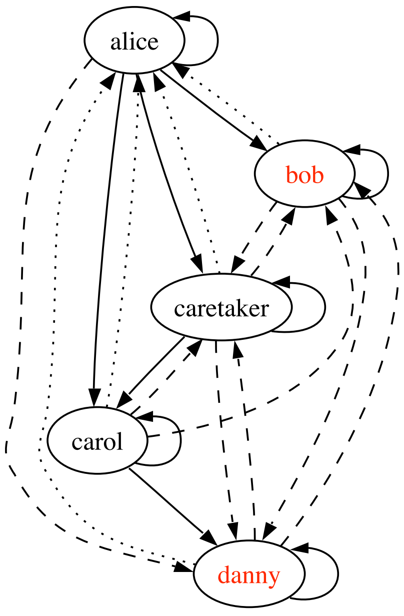


Figure 8.7: The graph for the first three solutions found

SCOLLAR finds 3 solutions before timing out after 30 seconds, imposing restrictions on 31 behavior facts, that are presented in table 8.14. The table is split up in restrictions for `alice` (top part) and restrictions for `carol` (bottom part).

The following facts are interesting to notice:

- In the first two solutions, neither `bob` nor `danny` get access to `alice`. Therefore `alice`’s behavior is less restricted.
- For the same reason, in the first two solutions `carol` is allowed to return herself to invokers that prove to here that they have access to `alice`: `carol:may.returnFor(alice,carol)`.

Table 8.14: The first three solutions for `alice`'s and `carol`'s behavior restrictions

Solutions	①	②	③
alice:may.sendTo(bob,alice)	**0**	**0**	1
alice:may.sendTo(bob,carol)	**0**	**0**	**0**
alice:may.sendTo(bob,danny)	**0**	**0**	**0**
alice:may.sendTo(caretaker,alice)	1	**0**	1
alice:may.sendTo(carol,alice)	1	**0**	1
alice:may.sendTo(danny,alice)	**0**	**0**	1
alice:may.sendTo(danny,carol)	**0**	**0**	**0**
alice:may.return(carol)	1	1	**0**
alice:may.return(danny)	1	1	**0**
alice:may.returnFor0(carol)	1	1	**0**
alice:may.returnFor0(danny)	1	1	**0**
alice:may.returnFor(alice,carol)	_	_	**0**
alice:may.returnFor(bob,carol)	_	_	**0**
alice:may.returnFor(caretaker,carol)	_	_	**0**
alice:may.returnFor(danny,carol)	1	1	**0**
carol:may.sendTo(bob,alice)	**0**	1	1
carol:may.sendTo(bob,carol)	**0**	**0**	**0**
carol:may.sendTo(bob,danny)	**0**	**0**	**0**
carol:may.sendTo(danny,alice)	**0**	1	1
carol:may.sendTo(danny,carol)	**0**	**0**	**0**
carol:may.return(alice)	**0**	1	1
carol:may.return(carol)	**0**	**0**	**0**
carol:may.returnFor0(alice)	**0**	1	1
carol:may.returnFor0(carol)	**0**	**0**	**0**
carol:may.returnFor(alice,carol)	1	1	**0**
carol:may.returnFor(bob,alice)	**0**	1	1
carol:may.returnFor(bob,carol)	**0**	**0**	**0**
carol:may.returnFor(caretaker,alice)	**0**	1	1
carol:may.returnFor(caretaker,carol)	**0**	**0**	**0**
carol:may.returnFor(danny,alice)	**0**	1	1
carol:may.returnFor(danny,carol)	**0**	**0**	**0**

8.3 Confinement

The confinement problem was described by Lampson in [Lam73]. Saltzer and Schroeder [SS73] defined the confinement problem as follows:

"Allowing a borrowed program to have access to data, while ensuring that the program cannot release the information."

In the context of this work, taking into account the difference between permission and authority, we adapt this definition to:

Definition 31. *"Giving an unknown subject authority, while ensuring that this does not lead to the propagation of that authority beyond a predefined perimeter."*

SCOLLAR's raison d'être is all about a generalized notion of confinement: to find ways to restrict the permissions of the subjects and the behavior of the relied-upon subjects in a pattern, that prevent authority from propagating beyond the limits set by a safety policy.

SCOLLAR may come up with different solutions for different situations. There may also be situations where no solution is found. We use SCOLLAR to find, investigate, and understand precisely defined patterns of interaction that can be applied over again in situations that are similar enough to be recognizable instances of the pattern. The confinement perimeter is defined by the safety properties in that pattern.

We can also use SCOLL to express a strict interpretation of the confinement problem (Definition 31) that minimizes the theoretically feasible confinement perimeter. The membrane pattern expresses the strictest sensible interpretation of this problem: subjects whose behavior is unknown but whose permissions do not include access to each other, should be allowed to communicate indirectly, but should never get access to each other.

We cannot interpret the confinement problem more strictly that this, without losing its general relevance for all sorts of authority. Suppose that a system is able to prevent two unknown subjects that do have access to each other (and thus are allowed and able to use each other) from delegating (portable) authority to each other. Even then, the system cannot prevent them from using their authority on each other's behalf. As we have seen in section 8.1, a subject's intention is a matter of its behavior, and is by definition not controlled for unknown subjects.

8.3.1 Inescapable Interposition: The Membrane Pattern

The membrane pattern relies on the behavior of subjects that are inter-positioned between the two parties that have to be kept permission-separated. In that sense it is similar to the caretaker pattern, where `bob` and `carol` were permission-separated by the `caretaker`.

Contrary to the caretaker pattern that relies on the restricted behavior of `carol`, the membrane pattern does not depend on behavior restrictions in either of the permission-separated subjects. Instead, the inter-positioned subjects must make sure themselves that they always stay inter-positioned.

This can be achieved by relied-upon proxies that wrap every argument into a proxy, before passing it on to the subject on the other side of the confinement barrier (membrane). Because this proxy also has this "wrapping" behavior, similar to the behavior of the inter-positioned subject that created the proxy, the permission-separated subjects will always remain permissions-separated. By being consistently inter-positioned between the confined subjects, these proxies will have complete control over the authority the confined subjects can propagate to each other.

The membrane pattern was already described in section 6.9 to give an example of how subject aggregation can be used in practice. The confined subjects each model one of the originally permission-separated entities together with its potential offspring. Because these subjects are modeled with completely unrestricted behavior, and each of them is given access to itself, the aggregation is safe.

The inter-positioned subjects (proxies) each model an originally inter-positioned entity, and all other proxying entities created at runtime that proxy for a confined entity modeled by the same confined subject. Because the inter-positioned subjects all have the same behavior, the aggregation is safe. The proxies get access to each other because of their mutual `child` relation that models the aggregation.

The subjects to be kept permission-separated are `alice` and `bob`. The inter-positioned proxies upon whose behavior we rely are `proxyAlice` and `proxyBob`.

Whereas the actual implementation of the pattern may provide a challenge to avoid an explosion of inter-positioned proxies, its representation is SCOLL is very simple. Figure 8.8 shows the membrane pattern in SCOLL. It tells SCOLLAR to look for all necessary restrictions in the behavior of both authority-separated subjects: `alice` and `bob`.

The Solution

Figure 8.9 shows the single solution SCOLLAR found to the pattern. No restrictions have to be put on neither of the confined subjects `alice` and `bob`. Both subjects are kept permission-separated as can be seen from the graph. This means that the pattern is safe for use with untrusted subjects.

Applying the Membrane Pattern

The membrane pattern shows the advantages of relying on the behavior of subjects to impose safety requirements. The pattern can be applied for different kinds of confinement assignments, simply by adapting the behavior of the relied-upon proxies. For instance, if we make sure that all proxies are connected to a controller subject, and that all proxies respond to a revoke command from that controller by stopping all collaborative behavior, we have a revocation pattern in which all authority that was ever provided to the confined subject(s) can be revoked, without relying on the restrictions in any other subject than the ones especially provided for that purpose.

Since the membrane pattern does not rely on any restrictions in the behavior of the entities on either side of the barrier, it is applicable in symmetrical conditions where two equally important worlds have to be permission-separated, as well as in asymmetrical conditions where the part inside the membrane is to be protected from the world outside.

8.4 Delegation Considered Harmful for Confinement?

In 1984 Boebert claimed in a short note [Boe84] that "unmodified" capability machines are unable to enforce the $*$-property, because: "the right to exercise access carries with it the right to propagate access".

```
declare
  permission:  access/2
  behavior:  may.sendTo/3 may.getFrom/2 may.return/2 may.endow/3
   may.receive/1 may.returnFor/3 may.returnFor0/2
  knowledge:  did.sendTo/3 did.getFrom/3 did.return/2
   did.receive/2 did.returnFor/3 did.returnFor0/2
   did.getFromFor/4 did.getFromFor0/3 was.endowedWith/2
system
  B:may.receive() A:may.sendTo(B,X) access(A,B)
   access(A,X) => A:did.sendTo(B X);
  A:did.sendTo(B X) => B:did.getFrom(X);
  B:did.getFrom(X) => access(B,X);
  A:may.getFrom(B) B:may.return(X) access(A,B)
   access(B,X) => A:did.getFrom(B,X);
  A:did.getFrom(B,X) => access(A,X);
  A:did.getFrom(B,X) => B:did.return(X);
  B:may.returnFor(X,Y) A:may.sendTo(B,X) B:may.receive()
   access(B,Y) access(A,X) access(A,B)
   => A:did.getFromFor(B,X,Y);
  A:did.getFromFor(B,X,Y) => B:did.returnFor(X,Y);
  A:did.getFromFor(B,X,Y) => access(A,Y);
  B:did.returnFor(X,Y) => access(B,X);
  access(A,B) access(B,Y)
   A:may.getFrom(B) B:may.returnFor0(Y)
   => B:did.returnFor0(Y) A:did.getFromFor0(B,Y);
  B:may.return(Y)
   => B:may.returnFor(X,Y) B:may.returnFor0(Y);
  A:did.getFromFor0(B,Y) => A:did.getFrom(B,Y);
  B:did.returnFor0(Y) => B:did.return(Y);
  child(P,C) => access(P,C) myChild(P,C);
  child(P,C) access(P,X) P:may.endow(C,X)
   => C:was.endowedWith(X);
behavior
  MEMBRANE { was.endowedWith(T) => target(T);
  => may.receive();
  target(T) => may.getFrom(T);
  did.getFrom(X) myChild(C) target(T) =>
   => may.endow(C,X) may.sendTo(T,C);
  did.getFrom(P,X) myChild(C) => may.endow(C,X) may.return(C);}
  MINIMAL {}
subject alice:MINIMAL ?bob:MINIMAL
  proxyAlice:MEMBRANE proxyBob:MEMBRANE
config
  access(alice,alice) access(bob,bob)
  access(proxyAlice,proxyAlice)
  access(proxyBob,proxyBob) access(alice,proxyBob)
  access(bob,proxyAlice) access(proxyAlice,alice)
  access(proxyBob,bob) target(proxyAlice alice)
  target(proxyBob bob) alice:child(alice,alice) child(bob,bob)
  child(proxyAlice,proxyBob) child(proxyBob,proxyAlice)
goal
  !access(bob,alice !access(alice,bob)
```

Figure 8.8: The membrane pattern in SCOLL.

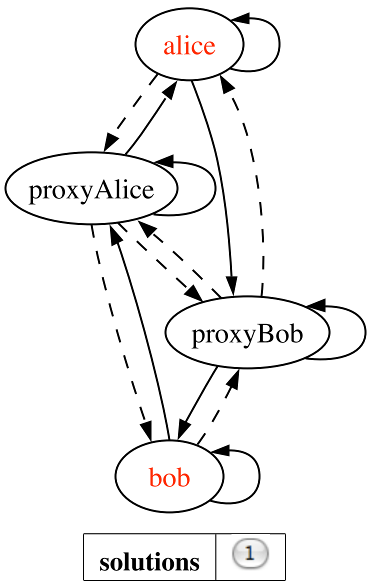


Figure 8.9: The unique solution to the membrane pattern: no restrictions are necessary. The pattern is safe for use with unknown subjects `alice` and `bob`.

Miller claims to have laid this problem to rest in chapter 11 of [Mil06b]. The only reason we dig it back up here is: to analyze it in terms of SCOLL patterns and to learn from the confusion this note has caused until very recently, in papers with a considerable impact, such as [KL87, Gon89, WBDF97, HKN05].

8.4.1 The $*$-Property

The $*$-*property* (star-property) is multi-level security policy in which all subjects get a clearance level and all objects get a confidentiality level. There is a partial order between the clearance levels, a one-to-one correspondence between clearance levels and confidentiality levels, and a corresponding partial order between the confidentiality levels.

The $*$-property states: agents should be able to write to all objects with a confidentiality level above (and including) the agent's corresponding level of clearance and to read from all levels below (and including) the agent's corresponding level of clearance, but no agent should be able to write strictly below that level, or read strictly above that level.

8.4.2 Boebert's Proof

Boebert claims that a policy that hands capabilities to agents corresponding to their clearance level, cannot guarantee the $*$-property. His proof relies on the interpretation of : "The right to exercise access carries with it the right to propagate access" as: "The permission to exercise access carries with it the *authority* to propagate access".

figure 8.10 shows a SCOLL pattern that models his proof set-up in [Boe84], using system rules that model this interpretation:

Read authority implies take authority : If A has **read** authority to B, then A has the *permission* to propagate all of B's capability's to A.

Write authority implies grant authority : If A has **write** authority to B, then A has the *permission* to propagate all of A's capability's to B.

We will call this the "implied authority" interpretation of capability delegation. The user is referred to Section 3.4 for a classification based on the relations between authority and permission.

The config part expresses the read and write permissions conform with the Multi Level Security policy. The goal express the safety property for the pattern Boebert used in his proof: highAgent should not have the authority to write to lowFile.

We use SCOLLAR in its first operation mode (See "Fixpoint Computation Mode" in section 7.2.1). Since we have no subject for which we want to maximize its permitted behavior, it will not matter whether we push the *minimal fixpoint* or the *maximal fixpoint* button to get the result.

The Results

Table 8.15 shows an extraction of the fixpoint calculation results, containing knowledge facts of lowAgent and highAgent. It clearly indicates that the safety property is violated, as Boebert predicted. What happened can be derived from the knowledge facts in that table:

The result of the fixpoint calculation shows that Boebert's further reasoning was sound: the safety property !highAgent:did.write(lowFile) is violated.

1. lowAgent:did.sendWriteAccessTo(lowfile,lowFile) : lowAgent did grant his write access to lowFile.

2. highAgent:did.getWriteAccessFrom(lowfile,lowFile) : highAgent took write access from lowFile.

3. highAgent:did.write(lowfile) : highAgent used his write access to lowFile.

In the next section we will present the alternative interpretation of delegation in capability systems: "The right to exercise access carries with it the right to propagate access" will no longer be interpreted as: "The permission to exercise access carries with it the *authority* to delegate access", but as: "The permission to exercise access carries with it the *permission* to delegate access".

Notice that this is the only change with the original interpretation: authority has been replaced by permission. This is a very important difference, certainly in capability systems, where no lower bound is set for the authority that is guaranteed by a permission. (See section 3.4).

8.4.3 A Closer Look at Delegation in Capability Systems

The system rules in the SCOLL program of figure 8.10 are not suitably refined to model the original capabilities that were proposed by Dennis and Van Horn (DVH) [DH65].

```
declare
   permission:  readAccess/2 writeAccess/2
   behavior:  may.sendTo/3 may.getFrom/2 may.read/2
      may.getReadAccessFrom/3 may.getWriteAccesFrom/3
      may.write/2 may.sendReadAccessTo/3
      may.sendWriteAccesTo/3
   knowledge:  did.read/2 did.write/2
      did.getReadAccessFrom/3 did.getWriteAccessFrom/3
      did.sendReadAccessTo/3 did.sendWriteAccessTo/3
system
   /*read and write rules*/
   readAccess(A,B) A:may.readB) => A:did.read(B);
   writeAccess(A,B) A:may.writeB) => A:did.write(B);
   /*grant rules*/
   writeAccess(A,B) readAccess(A,X) A:may.sendTo(B,X)
    => readAccess(B,X) A:did.sendReadAccessTo(B,X);
   writeAccess(A,B) writeAccess(A,X) A:may.sendTo(B,X)
   => writeAccess(B,X) A:did.sendWriteAccessTo:(B,X);
   /*take rules*/
   readAccess(A,B) readAccess(B,X)
   A:may.getReadAccessFrom(B,X)
   => readAccess(A,X) A:did.getReadAccessFrom(B,X);
   readAccess(A,B) writeAccess(B,X)
   A:may.getWriteAccessFrom(B,X)
    => writeAccess(A,X) A:did.getWriteAccessFrom(B,X);
behavior
   UNKNOWN { => may.read(X) may.write(X)
    may.getReadAccessFrom(A) may.sendReadAccessTo(A,X)
    may.getWriteAccessFrom(A)may.sendWriteAccessTo(A,X);}
   MINIMAL {}
subject
   highAgent:UNKNOWN lowAgent:UNKNOWN lowFile:MINIMAL
config
   readAccess(lowAgent,lowFile)
   writeAccess(lowAgent,lowFile)
   readAccess(highAgent,lowFile)
goal
   !highAgent:did.write(lowFile)
```

Figure 8.10: The SCOLL pattern expressing the “implied authority” interpretation of capability delegation.

Table 8.15: The knowledge facts from the fixpoint calculation

highAgent	highAgent	lowAgent	lowFile
readAccess(highAgent,_)	0	0	**1**
writeAccess(highAgent,_)	0	0	**1**
highAgent:did.read(_)	0	0	**1**
highAgent:did.write(_)	0	0	**1**
highAgent:did.getReadAccessFrom(highAgent,_)	0	0	0
highAgent:did.getReadAccessFrom(lowAgent,_)	0	0	0
highAgent:did.getReadAccessFrom(lowFile,_)	0	0	**1**
highAgent:did.getWriteAccessFrom(highAgent,_)	0	0	0
highAgent:did.getWriteAccessFrom(lowAgent,_)	0	0	0
highAgent:did.getWriteAccessFrom(lowFile,_)	0	0	**1**
highAgent:did.sendReadAccessTo(highAgent,_)	0	0	0
highAgent:did.sendReadAccessTo(lowAgent,_)	0	0	0
highAgent:did.sendReadAccessTo(lowFile,_)	0	0	**1**
highAgent:did.sendWriteAccessTo(highAgent,_)	0	0	0
highAgent:did.sendWriteAccessTo(lowAgent,_)	0	0	0
highAgent:did.sendWriteAccessTo(lowFile,_)	0	0	**1**

lowAgent	highAgent	lowAgent	lowFile
readAccess(lowAgent,_)	0	0	**1**
writeAccess(lowAgent,_)	0	0	**1**
lowAgent:did.read(_)	0	0	**1**
lowAgent:did.write(_)	0	0	**1**
lowAgent:did.getReadAccessFrom(highAgent,_)	0	0	0
lowAgent:did.getReadAccessFrom(lowAgent,_)	0	0	0
lowAgent:did.getReadAccessFrom(lowFile,_)	0	0	**1**
lowAgent:did.getWriteAccessFrom(highAgent,_)	0	0	0
lowAgent:did.getWriteAccessFrom(lowAgent,_)	0	0	0
lowAgent:did.getWriteAccessFrom(lowFile,_)	0	0	**1**
lowAgent:did.sendReadAccessTo(highAgent,_)	0	0	0
lowAgent:did.sendReadAccessTo(lowAgent,_)	0	0	0
lowAgent:did.sendReadAccessTo(lowFile,_)	0	0	**1**
lowAgent:did.sendWriteAccessTo(highAgent,_)	0	0	0
lowAgent:did.sendWriteAccessTo(lowAgent,_)	0	0	0
lowAgent:did.sendWriteAccessTo(lowFile,_)	0	0	**1**

They are a safe but coarse approximation of how permissions and authority propagate in capability systems. Because of that, the results of table 8.15 cannot be interpreted as a proof that an actual (unmodified) capability system cannot enforce the $*$-property.

An appropriately refined version of the system rules is easy to find, if we take the different kinds of capabilities DVH proposes into account. Miller suggests [Mil06a] that Boebert's system should have given appropriate *segment-capabilities* to the agents, designating data segment that are reserved for the corresponding level of confidentiality, instead of giving the agents *entry*-capabilities. The interested reader is referred to chapter 4 for a detailed account on these differences and to check that delegation in DVH indeed requires an *entry* capability.

Since DVH did not provide native *write-only-segment*-capabilities (write permission is always combined with read permission), we think it is equally valid to classify Boebert's conjecture as a confusion between authority and permissions.

In DVH, like in more recent capability systems, no permissions other than *access* to two subjects are needed to propagate the one subject to the other one. But the combination of these two permissions does not come with a guaranteed authority that the delegation will actually take place. It is the responder's behavior (in this case the behavior of `lowFile`) that will decide if it allows delegation or not. In DVH, like in most systems, a file is not capable of receiving or returning anything else but data. That means that, in our model the file subject can be trusted upon to be unwilling to return or receive any capabilities.

That leads us to a third possible explanation for the confusion: did Boebert confuse capabilities with data? At a certain level, capabilities are represented as bits in the system. However, DVH makes sure that these bits have no power outside their proper context: they only represent a capability when contained in the c-list of a process. (See chapter 4).

Figure 8.16 shows a plain capability based SCOLL pattern with Boebert's configuration of subjects. This time we make sure the system rules require the responder's consent, by re-introducing the `may.return` and `may.receive` behavior predicates.

The SCOLL pattern can be interpreted to be a correction for either of the three possible causes for Boebert's conjecture.

The results of the maximal fixpoint calculation are shown in figure 8.11 (access graph) and tables 8.17 to 8.22.

The graph shows no dashed arcs, which means that no access was propagated, even when the file was unrestricted. From the result, we learn that capability systems do not have to worry about their file's behavior restrictions: the read and write capabilities, when created as data diodes, are reliable enough. For the reader's reference, the complete results of an additional `maximal fixpoint` calculation of the same SCOLL pattern are provided in tables 8.17 until 8.22.

8.5 Reference Monitoring

This section provides some examples of how security strategies based on reference monitoring can be modeled in SCOLL.

8.5.1 Java's Sandbox

The Java virtual machine (JVM) can restrict the access external code has to the resources on the host computer. Usually, external code can use unlimited computation

Table 8.16: Boebert's pattern, using a sufficiently refined model of DVH capabilities

```
declare
   permission:  access/2
   behavior:  may.sendTo/3 may.getFrom/2 may.return/2
    may.receive/1 may.sendDataTo/2 may.getDataFrom/2
    may.returnData/1 may.receiveData/1
   knowledge:did.sendTo/3 did.getFrom/3 did.return/2
    did.receive/2 did.sendDataTo/2 did.getDataFrom/2
    did.returnData/1 did.receiveData/1
system
   B:may.receive() A:may.sendTo(B,X) access(A,B) access(A,X)
    => A:did.sendTo(B X);
   A:did.sendTo(B X) => B:did.getFrom(X);
   B:did.getFrom(X) => access(B,X);
   A:may.getFrom(B) B:may.return(X) access(A,B) access(B,X);
    => A:did.getFrom(B,X);
   A:did.getFrom(B,X) => access(A,X);
   A:did.getFrom(B,X) => B:did.return(X);
   B:may.receiveData() A:may.sendDataTo(B) access(A,B)
    => A:did.sendDataTo(B);
   A:did.sendDataTo(B) => B:did.receiveData();
   A:may.getDataFrom(B) B:may.returnData() access(A,B);
    => A:did.getDataFrom(B);
   A:did.getDataFrom(B) => B:did.returnData();
   A:did.getDataFrom(B) B:did.getDataFrom(C)
    => A:did.getDataFrom(C);
   A:did.sendDataTo(B) B:did.sendDataTo(C)
    => A:did.sendDataTo(C);
   A:did.sendDataTo(C) B:did.getDataFrom(C)
     => B:did.getDataFrom(A) A:did.sendDataTo(B);
behavior
   UNKNOWN { => may.receive() may.getFrom(A) may.return(X)
       may.sendTo(A,X) may.receiveData() may.getDataFrom(A)
       may.returnData() may.sendDataTo(A);}
   READCAP { isTarget(T) =>may.getDataFrom(T);
       did.getDataFrom(A)=>may.returnData();}
   WRITECAP { => may.receiveData();
       did.receiveData() isTarget(T) => may.sendDataTo(T);}
   MINIMAL {}
subject
   highAgent:UNKNOWN   lowAgent:UNKNOWN
   rCap1:READCAP   rCap2:READCAP   wCap:WRITECAP
   ?lowFile:MINIMAL
config
   access(highAgent,highAgent) access(highAgent,rCap1)
   access(rCap1,lowFile) access(lowAgent,lowAgent)
   access(lowAgent,rCap2) access(rCap2,lowFile)
   access(lowAgent,wCap) access(wCap,lowFile)
   rCap1:isTarget(lowFile)
   rCap2:isTarget(lowFile) wCap:isTarget(lowFile)
goal
   lowAgent:did.sendDataTo(lowFile)
   lowAgent:did.getDataFrom(lowFile)
   highAgent:did.getDataFrom(lowFile)
   !highAgent:did.sendDataTo(lowFile)
   !lowAgent:did.getDataFrom(highAgent)
   !highAgent:did.sendDataTo(lowAgent)
   !lowFile:did.getDataFrom(highAgent)
```

Table 8.17: The maximal fixpoint results for `highAgent`.

highAgent						
	high-Agent	low-Agent	rCap1	rCap2	wCap	low-File
access(highAgent,_)	**1**	0	**1**	0	0	0
highAgent:did.receiveData()	**1**					
highAgent:did.returnData()	**1**					
highAgent:did.getDataFrom(_)	**1**	**1**	**1**	**1**	**1**	**1**
highAgent:did.sendDataTo(_)	**1**	0	0	0	0	0
highAgent:did.getFrom(_)	**1**	0	**1**	0	0	0
highAgent:did.return(_)	**1**	0	**1**	0	0	0
highAgent:did.getFrom(highAgent,_)	**1**	0	**1**	0	0	0
highAgent:did.getFrom(lowAgent,_)	0	0	0	0	0	0
highAgent:did.getFrom(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
highAgent:did.getFrom(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
highAgent:did.getFrom(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
highAgent:did.getFrom(lowFile,_)	0	0	0	0	0	0
highAgent:did.sendTo(highAgent,_)	**1**	0	**1**	0	0	0
highAgent:did.sendTo(lowAgent,_)	0	0	0	0	0	0
highAgent:did.sendTo(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
highAgent:did.sendTo(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
highAgent:did.sendTo(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
highAgent:did.sendTo(lowFile,_)	0	0	0	0	0	0
highAgent:may.receive()	**1**					
highAgent:may.receiveData()	**1**					
highAgent:may.returnData()	**1**					
highAgent:may.getFrom(,_)	**1**	**1**	**1**	**1**	**1**	**1**
highAgent:may.getDataFrom(_)	**1**	**1**	**1**	**1**	**1**	**1**
highAgent:may.sendDataTo(_)	**1**	**1**	**1**	**1**	**1**	**1**
highAgent:may.return(_)	**1**	**1**	**1**	**1**	**1**	**1**
highAgent:may.sendTo(highAgent,_)	**1**	**1**	**1**	**1**	**1**	**1**
highAgent:may.sendTo(lowAgent,_)	**1**	**1**	**1**	**1**	**1**	**1**
highAgent:may.sendTo(rCap1,_)	**1**	**1**	**1**	**1**	**1**	**1**
highAgent:may.sendTo(rCap2,_)	**1**	**1**	**1**	**1**	**1**	**1**
highAgent:may.sendTo(wCap,_)	**1**	**1**	**1**	**1**	**1**	**1**
highAgent:may.sendTo(lowFile,_)	**1**	**1**	**1**	**1**	**1**	**1**

Table 8.18: The maximal fixpoint results for `lowAgent`.

lowAgent						
	high-Agent	low-Agent	rCap1	rCap2	wCap	low-File
access(lowAgent,_)	0	**1**	0	**1**	**1**	0
lowAgent:did.receiveData()	**1**					
lowAgent:did.returnData()	**1**					
lowAgent:did.getDataFrom(_)	0	**1**	0	**1**	**1**	**1**
lowAgent:did.sendDataTo(_)	**1**	**1**	**1**	**1**	**1**	**1**
lowAgent:did.getFrom(_)	0	**1**	0	**1**	**1**	0
lowAgent:did.return(_)	0	**1**	0	**1**	**1**	0
lowAgent:did.getFrom(highAgent,_)	0	0	0	0	0	0
lowAgent:did.getFrom(lowAgent,_)	0	**1**	0	**1**	**1**	0
lowAgent:did.getFrom(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowAgent:did.getFrom(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowAgent:did.getFrom(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowAgent:did.getFrom(lowFile,_)	0	0	0	0	0	0
lowAgent:did.sendTo(highAgent,_)	0	0	0	0	0	0
lowAgent:did.sendTo(lowAgent,_)	0	**1**	0	**1**	**1**	0
lowAgent:did.sendTo(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowAgent:did.sendTo(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowAgent:did.sendTo(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowAgent:did.sendTo(lowFile,_)	0	0	0	0	0	0
lowAgent:may.receive()	**1**					
lowAgent:may.receiveData()	**1**					
lowAgent:may.returnData()	**1**					
lowAgent:may.getFrom(_)	**1**	**1**	**1**	**1**	**1**	**1**
lowAgent:CollectData(_)	**1**	**1**	**1**	**1**	**1**	**1**
lowAgent:may.sendDataTo(_)	**1**	**1**	**1**	**1**	**1**	**1**
lowAgent:may.return(_)	**1**	**1**	**1**	**1**	**1**	**1**
lowAgent:may.sendTo(highAgent,_)	**1**	**1**	**1**	**1**	**1**	**1**
lowAgent:may.sendTo(lowAgent,_)	**1**	**1**	**1**	**1**	**1**	**1**
lowAgent:may.sendTo(rCap1,_)	**1**	**1**	**1**	**1**	**1**	**1**
lowAgent:may.sendTo(rCap2,_)	**1**	**1**	**1**	**1**	**1**	**1**
lowAgent:may.sendTo(wCap,_)	**1**	**1**	**1**	**1**	**1**	**1**
lowAgent:may.sendTo(lowFile,_)	**1**	**1**	**1**	**1**	**1**	**1**

Table 8.19: The maximal fixpoint results for `rCap1`.

rCap1						
	high-Agent	low-Agent	rCap1	rCap2	wCap	low-File
access(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**1**
rCap1:did.receiveData()	**1**					
rCap1:did.returnData()	**1**					
rCap1:did.getDataFrom(_)	0	**1**	0	**1**	**1**	**1**
rCap1:did.sendDataTo(_)	0	0	0	0	0	0
rCap1:did.getFrom(_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.return(_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.getFrom(highAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.getFrom(lowAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.getFrom(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.getFrom(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.getFrom(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.getFrom(lowFile,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.sendTo(highAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.sendTo(lowAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.sendTo(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.sendTo(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.sendTo(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:did.sendTo(lowFile,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:may.receive()	**0**					
rCap1:may.receiveData()	**0**					
rCap1:may.returnData()	**1**					
rCap1:may.getFrom(_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:may.getDataFrom(_)	**0**	**0**	**0**	**0**	**0**	**1**
rCap1:may.sendDataTo(_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:may.return(_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:may.sendTo(highAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:may.sendTo(lowAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:may.sendTo(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:may.sendTo(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:may.sendTo(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:may.sendTo(lowFile,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap1:isTarget(_)	**0**	**0**	**0**	**0**	**0**	**1**

Table 8.20: The maximal fixpoint results for `rCap2`.

rCap2						
	high-Agent	low-Agent	rCap1	rCap2	wCap	low-File
access(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**1**
rCap2:did.receiveData()	**1**					
rCap2;did.returnData()	**1**					
rCap2:did.getDataFrom(_)	0	**1**	0	**1**	**1**	**1**
rCap2:did.sendDataTo(_)	0	0	0	0	0	0
rCap2:did.getFrom(_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.return(_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.getFrom(highAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.getFrom(lowAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.getFrom(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.getFrom(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.getFrom(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.getFrom(lowFile,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.sendTo(highAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.sendTo(lowAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.sendTo(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.sendTo(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.sendTo(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:did.sendTo(lowFile,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:may.receive()	**0**					
rCap2:may.receiveData()	**0**					
rCap2:may.returnData()	**1**					
rCap2:Collect(_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:may.getDataFrom(_)	**0**	**0**	**0**	**0**	**0**	**1**
rCap2:may.sendDataTo(_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:may.return(_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:may.sendTo(highAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:may.sendTo(lowAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:may.sendTo(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:may.sendTo(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:may.sendTo(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:may.sendTo(lowFile,_)	**0**	**0**	**0**	**0**	**0**	**0**
rCap2:isTarget(_)	**0**	**0**	**0**	**0**	**0**	**1**

Table 8.21: The maximal fixpoint results for `wCap`.

wCap						
	high-Agent	low-Agent	rCap1	rCap2	wCap	low-File
access(wCap,_)	**0**	**0**	**0**	**0**	**0**	**1**
wCap:did.receiveData()	**1**					
wCap:did.returnData()	**1**					
wCap:did.getDataFrom(_)	0	0	0	0	0	0
wCap:did.sendDataTo(_)	**1**	**1**	**1**	**1**	**1**	**1**
wCap:did.getFrom(_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.return(_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.getFrom(highAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.getFrom(lowAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.getFrom(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.getFrom(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.getFrom(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.getFrom(lowFile,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.sendTo(highAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.sendTo(lowAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.sendTo(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.sendTo(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.sendTo(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:did.sendTo(lowFile,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:may.receive()	**0**					
wCap:may.receiveData()	**1**					
wCap:may.returnData()	**0**					
wCap:may.getFrom(_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:may.getDataFrom(_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:may.sendDataTo(_)	**0**	**0**	**0**	**0**	**0**	**1**
wCap:may.return(_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:may.sendTo(highAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:may.sendTo(lowAgent,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:may.sendTo(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:may.sendTo(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:may.sendTo(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:may.sendTo(lowFile,_)	**0**	**0**	**0**	**0**	**0**	**0**
wCap:isTarget(_)	**0**	**0**	**0**	**0**	**0**	**1**

Table 8.22: The maximal fixpoint results for `lowFile`.

lowFile						
	high-Agent	low-Agent	rCap1	rCap2	wCap	low-File
access(lowFile,_)	0	0	0	0	0	0
lowFile:did.receiveData()	**1**					
lowFile:did.returnData()	**1**					
lowFile:did.getDataFrom(_)	0	0	0	0	0	0
lowFile:did.sendDataTo(_)	0	0	0	0	0	0
lowFile:did.getFrom(_)	0	0	0	0	0	0
lowFile:did.return(_)	0	0	0	0	0	0
lowFile:did.getFrom(highAgent,_)	0	0	0	0	0	0
lowFile:did.getFrom(lowAgent,_)	0	0	0	0	0	0
lowFile:did.getFrom(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowFile:did.getFrom(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowFile:did.getFrom(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowFile:did.getFrom(lowFile,_)	0	0	0	0	0	0
lowFile:did.sendTo(highAgent,_)	0	0	0	0	0	0
lowFile:did.sendTo(lowAgent,_)	0	0	0	0	0	0
lowFile:did.sendTo(rCap1,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowFile:did.sendTo(rCap2,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowFile:did.sendTo(wCap,_)	**0**	**0**	**0**	**0**	**0**	**0**
lowFile:did.sendTo(lowFile,_)	0	0	0	0	0	0
lowFile:may.receive()	textbf1					
lowFile:may.receiveData()	**1**					
lowFile:may.returnData()	**1**					
lowFile:may.getFrom(_)	**1**	**1**	**1**	**1**	**1**	**1**
lowFile:may.getDataFrom(_)	**1**	**1**	**1**	**1**	**1**	**1**
lowFile:may.sendDataTo(_)	**1**	**1**	**1**	**1**	**1**	**1**
lowFile:may.return(_)	**1**	**1**	**1**	**1**	**1**	**1**
lowFile:may.sendTo(highAgent,_)	**1**	**1**	**1**	**1**	**1**	**1**
lowFile:may.sendTo(lowAgent,_)	**1**	**1**	**1**	**1**	**1**	**1**
lowFile:may.sendTo(rCap1,_)	**1**	**1**	**1**	**1**	**1**	**1**
lowFile:may.sendTo(rCap2,_)	**1**	**1**	**1**	**1**	**1**	**1**
lowFile:may.sendTo(wCap,_)	**1**	**1**	**1**	**1**	**1**	**1**
lowFile:may.sendTo(lowFile,_)	**1**	**1**	**1**	**1**	**1**	**1**

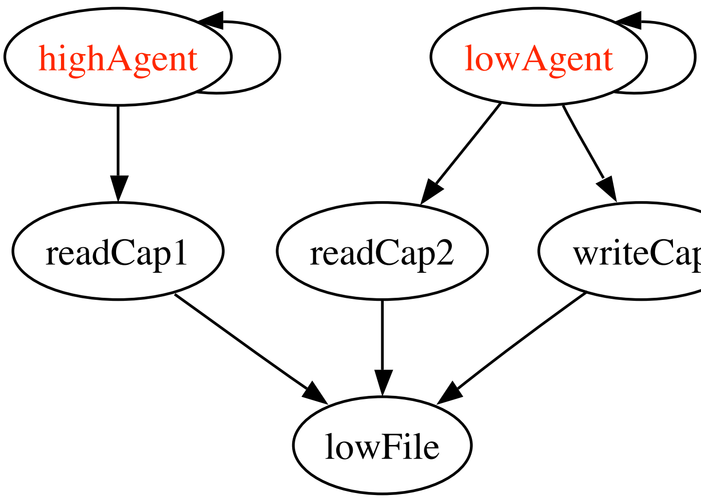


Figure 8.11: The resulting access graph

time and memory on the host computer, and is allowed to access the web server from which the code was downloaded. The JVM has a built-in reference monitor: the Java Security Manager (JSM). The JSM intercepts method calls, and prevents their execution when necessary by throwing a security exception.

The sandbox reference monitor knows two principals: `trusted` and `untrusted`. It safely approximates the principal on whose behalf a security-sensitive method is called, by assuming that the method is called on behalf of `untrusted` as soon as one frame in the call stack leading to that call originates from `untrusted`. For that purpose, stack frames are adorned with the class-loader of the class whose method is invoked in the frame. The `trusted` and `untrusted` principals can be recognized by that class-loader.

Contrary to capability-based security policies, enforcing the sandbox policy does not require secure programming: it is the responsibility of the JSM. The programmer relies on it to work as specified. Of course, the behavior of the relied-upon entities can still be programmed to implement further confinement of authority. Section 8.5.3 will show why this is still necessary. Since we don't need to model behavior, we can model the policy in SCOLL using only permissions and system rules.

A SCOLL pattern for a simplified sandbox is presented in figure 8.12. It models a sandbox policy that does not allow applets to access the web, not even their own download site. We will release this restriction later.

Subjects and Predicates

We use the unary predicate `trusted`, to indicate entities that were loaded from the system's main class loader: the class loader that only loads local code, not remote code (applets). Two aggregated subjects will suffice to model all trusted entities: `alice` and `carol`. A third aggregated subject, `bob`, will model all entities loaded from remote (untrusted) code. Notice that we did not use the term "relied-upon" in this case, to reflect the fact that we do not rely on behavior, even if it is "trusted". We rely only on the Java Security Manager.

```
declare
   permission:  safe/1 trusted/1 call/2
   behavior:
   knowledge:
system
   /* all entities are permitted to call safe entities */
   safe(X) => call(A,X);
   /* trusted entities can call all entities */
   trusted(A) => call(A,X);
behavior
subject
   alice
   bob
   carol
config
   trusted(alice)
   trusted(carol)
   safe(carol)
goal
   !call(bob,alice)
```

Figure 8.12: The SCOLL pattern expressing Java's sandbox, excluding the "call home" policy.

We use the unary predicate `safe` to indicate software entities that can be used at no risk, because they do not represent a protected resource. Untrusted subjects, like `bob`, are considered to be safe: they cannot represent any protected resources.

Both permissions are assumed to be static: safe entities can never become unsafe, trusted entities can never become untrusted, and vice versa.

The binary predicate `call` expresses the permission of a subject to call another subject, either directly or indirectly. The `call` permission is different from the `access` permission we encountered in capability systems in three aspects:

1. The `access` permission in capability systems only regulates direct invocation, whereas the `call` permission regulates indirect invocation as well.

2. Like all permissions in pure reference monitoring strategies, `call` permissions are not usable as references (designation) to the invokable subject. Even as Java is a memory safe language, and references are unforgeabe, references cannot be used as permissions and permissions cannot be used as references in the sandbox policy.

3. Like all permissions in pure reference monitoring strategies, `call` permissions do not guarantee the opportunity to use them. To use a permission, one may sometimes need a reference to the entity to be invoked directly, or to an entity that has a reference to the entity to be invoked indirectly, and so on. But because references are not explicitly modeled here, we must assume that the opportunity will always be present anyway, to make sure our model is a safe approximation of the real program.

The predicates `trusted`, `safe`, and `call` are all permission predicates: the entities do not have to be aware of them. Because the entities are not relied upon to restrict the use of their permissions in any way, we do not need behavior predicates.

System Rules

```
/* all entities are permitted to call safe entities */
safe(X) => call(A,X);
/* trusted entities can call all entities */
trusted(A) => call(A,X);
```

The system rules simply state the rules that are imposed by the reference monitor at runtime upon inspection of the call stack: for subject `A` to be allowed to invoke subject `X` directly or indirectly, (`call(A,X)`), `A` must be `trusted` or `X` must be `safe`.

Note that the `call` permission is only transitive in situations where the caller is trusted or where the callee is safe. The system rules to express this restricted transitivity would therefore look like this:

```
call(A,B) call(B,X) safe(X) => call(A,X);

trusted(A) call(A,B) call(B,X) => call(A,X);
```

Both situations are already covered by the simpler system rules.

Interpretation of the SCOLL Results

The fixpoint calculation results are shown in figure 8.13. The graph shows the `call` permissions that can be derived from the system rules. As expected, `bob` has no permission to call `alice` directly or indirectly. Because all entities are aggregated according to their static permissions `safe` and `trusted` the pattern safely approximates all Java programs, and the result constitutes a real proof about the safety property: no untrusted entities modeled by `bob` will ever have a permission to call any unsafe entities modeled by `alice`, neither directly nor indirectly.

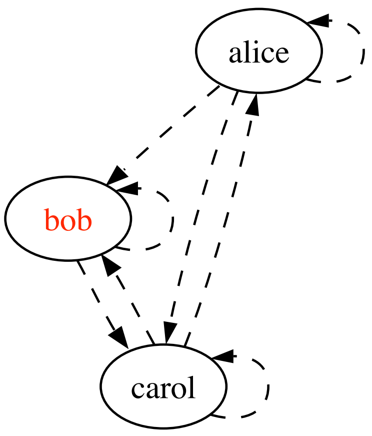
Note that the aggregation strategy we use to arrive at this simple model is not fitted for proving the safety of the "call home" permission. Different untrusted entities may be loaded from different sites, and are allowed to access their own download site but not each other's.

Safe aggregation dictates that, if we choose to aggregate every entity that provides access to an applet download site into the same subject, we have to give the `safe` permission to that subject, because it is safe to at least one untrusted applet. All trusted entities that provide access to an applet download site will thus be aggregated into `carol`, not `alice`.

Our pattern is then still a safe approximation, but not refined enough to prove that untrusted entities cannot access each other's download site. To solve this problem, we will refine the `safe` permission in section 8.5.2.

8.5.2 Allowing Applets to Call Home

To model the permissions for applets to call their own download site, we will single out one download site, and aggregate all entities of the applets downloaded from that site into `bob`. All entities in the other applets will be aggregated into a new subject `other`. The trusted entities that give access to these sites will be modeled as `bobSite` and `otherSite`. The entities that are completely safe are modeled as `carol`. The subject `otherSite` aggregates all trusted entities that give access to any other download site.



alice			
	alice	bob	carol
safe(alice)	0		
trusted(alice)	1		
call(alice,_)	1	1	1

bob			
	alice	bob	carol
safe(bob)	1		
trusted(bob)	0		
call(bob,_)	0	1	1

carol			
	alice	bob	carol
safe(carol)	1		
trusted(carol)	1		
call(carol,_)	1	1	1

Figure 8.13: The result of the fixpoint calculation for the Java sandbox.

Using the refinement technique explained in section 5.5, we refine the unary predicate `safe` into the binary predicate `safeFor`.

The refined pattern is shown in figure 8.14.

```
declare
   permission:  safe/1 safeFor/2 trusted/1 call/2
   behavior:
   knowledge:
system
   safe(X) => safeFor(A,X);
   safeFor(A,X) => call(A,X);
   trusted(A) => call(A,X);
behavior
subject
   alice
   bob
   other
   bobSite
   otherSite
   carol
config
   trusted(alice) trusted(carol)
   safe(carol) safe(bob) safe(other)
   safeFor(bob,bobSite) safeFor(other,otherSite)
goal
   !call(bob,alice)
   !call(other,alice)
   !call(bob,otherSite)
   !call(other,bobSite)
```

Figure 8.14: The SCOLL pattern expressing Java's sandbox, including the "call home" policy.

The system rules are adapted to the refinement as usual. The subject part of the SCOLL pattern is extended with the extra subjects, and the config part uses the refined `safeFor` permissions to indicate that `bobSite` is safe to use for `bob`, and `otherSite` is safe to use for `other`. The goals are adapted accordingly to indicate what safety properties we expect to hold.

The graph depicting the `call` permissions is presented in figure 8.15. The detailed results are shown in figure 8.16.

Notice that `bob` cannot invoke `otherSite` and `other` cannot invoke `bobSite`, while each of them is allowed access to their own site. This means that the safety properties are guaranteed as far as the entities of the applets that originate from `bob`'s site are concerned. Because no other assumptions were made about `bob` and `bobSite`, a similar aggregation model can be made for every site from which applets can be downloaded to prove the safety properties for the applets originating from that site. From this observation follows directly that the safety properties are valid for all applets.

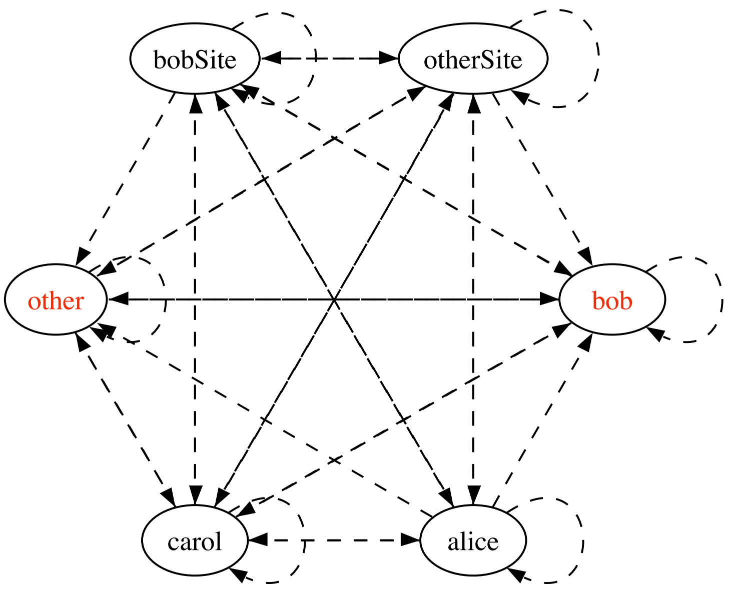


Figure 8.15: The `call` permission graph

8.5.3 Java's Sandbox and Authority Control

The goal of the sandbox strategy is to prevent direct or indirect invocation of unsafe entities by untrusted entities. Of course, we have seen in section 1.3.3 already (Figure 1.2), authority flow is not restricted to a single, direct or indirect, invocation :

- In situations where the trusted unsafe entity `alice` invokes the untrusted entity `bob`, `bob` can have authority over `alice`, for instance if `alice` would probe `bob` to get instructions from him about what to do when `bob`'s method is no longer on the call stack. If that is to be prevented, we should restrict `alice`'s behavior, and rely on it that she does not invoke `bob` with the intention to ask instructions and execute them.

- Even when `bob`'s and `alice`'s methods are never on the call stack at the same time, there are ways for `bob` to causally influence `alice`. For instance, a trusted and safe third party, like `carol`, may get instructions from `bob` about what `alice` should do. Later, when `bob`'s method has disappeared from the stack, `alice` could get these instruction from `carol`.

Therefore, while the sandbox strategy is very effective to limit the permissions of untrusted entities, it cannot by itself guarantee authority confinement. Permission restrictions alone are not sufficient to restrict the authority of untrusted entities to cause the same effects of having the permission.

Most criticism on the sandbox model focusses on its lack of expressive power: its inflexible all-or-nothing policy. However, since safety is the main goal, it is even more important to acknowledge the fact that, even with the most drastical form of

alice						
	alice	bob	other	bobSite	otherSite	carol
safe(alice)	0					
trusted(alice)	1					
call(alice,_)	1	1	1	1	1	1
safeFor(alice,_)	0	1	1	0	0	1

bob						
	alice	bob	other	bobSite	otherSite	carol
safe(bob)	1					
trusted(bob)	0					
call(bob,_)	0	1	1	1	0	1
safeFor(bob,_)	0	1	1	1	0	1

other						
	alice	bob	other	bobSite	otherSite	carol
safe(other)	1					
trusted(other)	0					
call(other,_)	0	1	1	0	1	1
safeFor(other,_)	0	1	1	0	1	1

bobSite						
	alice	bob	other	bobSite	otherSite	carol
safe(bobSite)	0					
trusted(bobSite)	1					
call(bobSite,_)	1	1	1	1	1	1
safeFor(bobSite,_)	0	1	1	0	0	1

otherSite						
	alice	bob	other	bobSite	otherSite	carol
safe(otherSite)	0					
trusted(otherSite)	1					
call(otherSite,_)	1	1	1	1	1	1
safeFor(otherSite,_)	0	1	1	0	0	1
carol						
	alice	bob	other	bobSite	otherSite	carol
safe(carol)	1					
trusted(carol)	1					
call(carol,_)	1	1	1	1	1	1
safeFor(carol,_)	0	1	1	0	0	1

Figure 8.16: The result of the fixpoint calculation for the “call home” sandbox policy.

sandboxing, we still have to rely on the behavior of the trusted entities to restrict the authority.

We therefore have to perform a behavior-based analysis of eventual authority, to be sure that the required safety properties regarding the authority of the untrusted entities hold. The results of such an analysis will depend on the actual Java program that is being modeled in SCOLL and on the precision the model.

Notice also that it may not always be appropriate to treat all locally loaded classes and their instances as `trusted`. In situations where this is not appropriate, sandboxing starts from an approximation that is not only crude but also unsafe.

8.5.4 Stack Walking

We have discussed stack walking briefly in section 8.1.2, and we will now express a simple example of this strategy in SCOLL, to take the discussion to a more formal level. At this stage we want to remind the reader once more that SCOLL was meant to express and examine behavior-based strategies, and that when safety in a pattern is broken, nothing can be inferred about the unsafety of the actual program: the safe approximation of the behavior can have been too coarse.

Contrary to sandbox strategies, stack walking is a behavior-based strategy: entities are relied upon to enable or disable their permissions on purpose.

We will not investigate if the behavior of enabling/disabling permissions can be proven to be a sufficiently restrictive strategy in general. That is left as interesting future work. In section 8.5.5 we will discuss some limitations of the strategy.

Another interesting topic for future research would be to check if accurate enabling and disabling behavior could be expressed separately from the functional part of the behavior. If such a division of concerns could be established, this would constitute a considerable improvement to the security maintainability of the software.

In "The Structure of Authority: Why Security is Not a Separable Concern" [MTS05] Miller et all. argue that building secure software is a design and development concern, and that POLA should be used as a major design principle during the building of secure software. They acknowledge that a degree of separation is possible in the sense of Dijkstra's suggestion: that we temporarily separate concerns as a conceptual aid for reasoning about complex systems [Dij82]. In "On the importance of the separation-of-concerns principle in secure software engineering" [WPJV03] De Win et all. shown that at least a limited form of separation of concerns for security aspects can become feasible, using new engineering techniques like aspect oriented programming.

To see if a SCOLL analysis can help to shed some light on this issue, we will use permission-enabling behavior to build non-confusable deputies. We modeled the capability-based strategy in section 8.1.3 where we showed that for a deputy entity to protect itself from being confused by its clients it suffices to require capabilities from the clients and to use only these capabilities for the client's purpose.

We provide a SCOLL model for a simple version of stack walking that only has permission enabling, not disabling. We show how to model this in SCOLL, and discuss the solutions found by SCOLLAR. We do not claim that our model is the most appropriate one. Different implementations of stack walking may express different flavors of the strategy. Our model corresponds to a simple variant that allows entities to put a permission on the stack while calling another entity. Doing so will dynamically delegate that permission to the entities that will be subsequently invoked. The reference monitor will check the stack when a resource is called, and only allow the call if it detects the permission on the stack. A non-monotonic version that also allows revocation of permissions that were put the stack is left as interesting future work.

Subjects and Predicates

The subjects will be the ones of the confused deputy example in section 8.1.3, except for the deputy itself. To express the deputy's intentional enabling of permissions, we propose to split up the deputy in two subjects. The first one is called `calcFacet` and represents the methods of the deputy object that performs the calculation, and writes the output of the calculation to the `client`'s file `cFile`. The second one is called `adminFacet` and represents the methods of the deputy object that read and write to the deputy's own file `dFile`. We therefore assume that no method performs both tasks.

A binary predicate `access` models the permission for the subjects to call, directly or indirectly, the protected resources `dFile` or `cFile`. A unary predicate `safe` indicates that a subject does not represent a protected resource.

We model two aspects of behavior. The binary behavior predicate `may.invoke` models the subject's intention to invoke another subject. A ternary predicate will refine this behavior: `may.invokeEnable` indicates that the invoker intends to put an access permission on the stack while calling a subject. The access permission will then be available to all entities that are called while this permission is on the stack.

The knowledge binary behavior predicate `did.invoke` indicates that the invoker succeeded in a direct invocation of another subject, be it a resource or not. The binary predicate `did.call` models successful direct or indirect calls. `did.invoke` refines `did.call`.

Figure 8.17 shows the complete SCOLL pattern.

System Rules

These are the system rules we propose, with a short explanation.

(1) `safe(X) A:may.invoke(X) => A:did.invoke(X);`
Non-resource entities can be invoked by everybody.

(2) `A:may.invokeEnable(B,X) => A:may.invoke(B);`
Behavior refinement rule.

(3) `A:did.invoke(X) => A:did.call(X);`
Knowledge refinement rule.

(4)
```
A:did.callEnable(B,X)
 => A:did.call(X) A:did.call(B) B:did.invoke(X);
```
Assistant rule.

(5) `A:did.call(B) B:did.call(C) => A:did.call(C);`
Transitivity rule.

(6)
```
access(A,X) A:may.invokeEnable(X,X)
 => A:did.callEnable(X,X) A:did.invoke(X);
```
Permission enabled direct invocation of resources.

(7)
```
access(A,X) A:may.invokeEnable(B,X) B:may.invoke(X)
 => A:did.callEnable(B,X) A:did.invoke(B);
```
Permission enabled indirect invocation of resources.

(8)
```
access(A,X) A:did.callEnable(B,X) B:did.call(C)
 C:may.invoke(X) => A:did.callEnable(C,X);
```
Transitivity for permission enabled indirect invocation of resources.

The Configuration

We give the `client` access permission to `cFile`. Both facets of the deputy get access permission to `dFile`.

In a first version, we search for the maximally permissive behavior of both deputy facets, given that the files can be relied upon to be completely passive entities that can only accept data and provide data.

In a second version, we search for similar solutions where `cFile` is completely unrestricted.

In a third version, we search for similar solutions where `dFile` is completely unrestricted.

Safety and Liveness Requirements

By preventing the untrusted entities `client` and `cFile` from directly invoking `dFile` directly and by preventing `calcFacet` from calling `dFile` directly or indirectly, we leave but one option for `client` to call `dFile`: indirectly via `adminFacet`. That is how the deputy intented to use that file.

To make sure that both facets can perform their job, we demand the remaining liveness possibilities:
`calcFacet:did.invoke(cFile)` and
`adminFacet.did.invoke(dFile)`.

Most important, we do not want `calcFacet` to call `dFile` directly or indirectly, because that could indicate that the deputy is confused.

The SCOLLAR Results: first version

In the first version, we search for the maximally permissive behavior of both deputy facets, given that the files can be relied upon to be completely passive entities that can

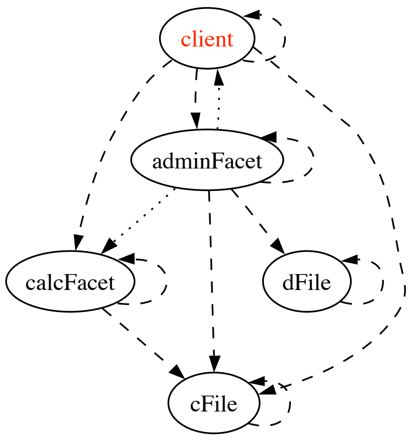
```
declare
    permission:  safe/1 access/2
    behavior:  may.invoke/2 may.invokeEnable/3
    knowledge:  did.invoke/2 did.call/2
system
    safe(X) A:may.invoke(X) => A:did.invoke(X);
    A:may.invokeEnable(B,X) =>A:may.invoke(B);
    A:did.invoke(X) => A:did.call(X);
    A:did.callEnable(B,X)
     => A:did.call(X) A:did.call(B) B:did.invoke(X);
    A:did.call(B) B:did.call(C) => A:did.call(C);
    access(A,X) A:may.invokeEnable(X,X)
     => A:did.callEnable(X,X) A:did.invoke(X);
    access(A,X) A:may.invokeEnable(B,X) B:may.invoke(X)
     => A:did.callEnable(B,X) A:did.invoke(B);
    access(A,X) A:did.callEnable(B,X) B:did.call(C)
     C:may.invoke(X) => A:did.callEnable(C,X);
behavior
    UNKNOWN { => may.invokeEnable(B,X);}
    MINIMAL {}
subject
    client:UNKNOWN
    ?adminFacet:MINIMAL
    ?adminFacet:MINIMAL
    cFile:MINIMAL dFile:MINIMAL
    /* second version:  cFile:UNKNOWN ?dFile:MINIMAL */
    /* third version:  ?cFile:MINIMAL dFile:UNKNOWN */
config
    safe(client) safe(adminFacet) safe(calcFacet)
    access(client,cFile)
    access(cFile,cFile) access(dFile,dFile)
    access(adminFacet,dFile) access(calcFacet,dFile)
goal
    client:did.call(dFile)
    adminFacet:did.invoke(dFile)
    calcFacet:did.invoke(cFile)
    !client:did.invoke(dFile)
    !cFile:did.invoke(dFile)
    !did.call(calcFacet,dFile)
```

Figure 8.17: A SCOLL pattern for non-confused deputies, using a strategy based on stack walking.

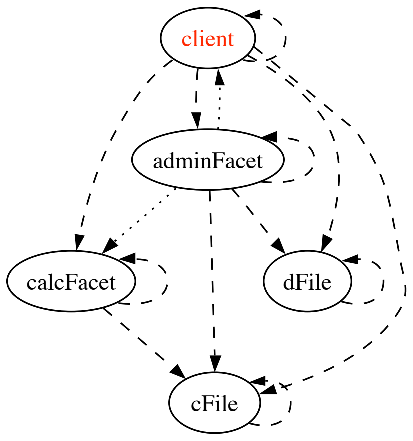
only accept data and provide data.

Four solutions are found. Figure 8.5.4 gives a graphical representation of the `did.invoke` relations (left) the `did.call` relations (right). Notice that only the `client` node is indicated in red, and not `dFile`. That is because we rely on `dFile` to be a passive file.

The graphs represent the knowledge reachable in all solutions by dashed arcs and the knowledge reachable in at least one solution with dotted arcs. The left graph shows clearly that `dFile` can only be called via `adminFacet`.



`did.invoke` graph



`did.call` graph

Table 8.23 lists the sets of behavior restrictions that are necessary and sufficient to comply with the requirements.

In no circumstances should `calcFacet` be programmed to invoke `client` or `adminFacet`, even without enabling any permission. Otherwise `calcFacet` may indirectly call `dFile`, which is what we regard to be confusion of the deputy. For the same reason, `calcFacet` is not allowed to invoke `dFile` while enabling its `dFile` permission.

The SCOLLAR Results: second and third versions

No solutions were found that allow unrestricted behavior for either `cFile` or `dFile`. Remember that the capability based approach did find a solution for that situation. However, in SCOLL, the absence of a safety proof is not a proof for the absence of safety. More refined safe approximations may exist that allow us to find a solution. Alternatively, we can still consider using a more expressive approach to stack walking that allows entities to also disable permissions on the stack.

8.5.5 Limitations of Stack walking Strategies

We have proven that a particular stack walking strategy can satisfy our specific safety and liveness requirements. Many other strategies are possible. Some of them may release the restrictions we encountered.

However, we should keep in mind that the safety requirements we expressed only intended to prevent the calculating facet of the deputy to use directly or indirectly the file that was only meant for the deputy's own administrative purpose. If we cannot rely

Table 8.23: The results for the stack walking example

Solutions	①	②	③	④
calcFacet:may.invoke(client)	**0**	**0**	**0**	**0**
calcFacet:may.invoke(dFile)	1	1	**0**	**0**
calcFacet:may.invoke(adminFacet)	**0**	**0**	**0**	**0**
calcFacet:may.invokeEnable(calcFacet,dFile)	**0**	**0**	1	1
calcFacet:may.invokeEnable(client,cFile)	**0**	**0**	**0**	**0**
calcFacet:may.invokeEnable(client,calcFacet)	**0**	**0**	**0**	**0**
calcFacet:may.invokeEnable(client,client)	**0**	**0**	**0**	**0**
calcFacet:may.invokeEnable(client,dFile)	**0**	**0**	**0**	**0**
calcFacet:may.invokeEnable(client,adminFacet)	**0**	**0**	**0**	**0**
calcFacet:may.invokeEnable(dFile,cFile)	1	1	**0**	**0**
calcFacet:may.invokeEnable(dFile,calcFacet)	1	1	**0**	**0**
calcFacet:may.invokeEnable(dFile,client)	1	1	**0**	**0**
calcFacet:may.invokeEnable(dFile,dFile)	**0**	**0**	**0**	**0**
calcFacet:may.invokeEnable(dFile,adminFacet)	1	1	**0**	**0**
calcFacet:may.invokeEnable(adminFacet,cFile)	**0**	**0**	**0**	**0**
calcFacet:may.invokeEnable(adminFacet,calcFacet)	**0**	**0**	**0**	**0**
calcFacet:may.invokeEnable(adminFacet,client)	**0**	**0**	**0**	**0**
calcFacet:may.invokeEnable(adminFacet,dFile)	**0**	**0**	**0**	**0**
calcFacet:may.invokeEnable(adminFacet,adminFacet)	**0**	**0**	**0**	**0**
adminFacet:may.invoke(calcFacet)	**0**	1	1	1
adminFacet:may.invoke(client)	**0**	1	**0**	1
adminFacet:may.invokeEnable(calcFacet,cFile)	**0**	1	1	1
adminFacet:may.invokeEnable(calcFacet,calcFacet)	**0**	1	1	1
adminFacet:may.invokeEnable(calcFacet,client)	**0**	1	1	1
adminFacet:may.invokeEnable(calcFacet,dFile)	**0**	**0**	1	1
adminFacet:may.invokeEnable(calcFacet,adminFacet)	**0**	1	1	1
adminFacet:may.invokeEnable(client,cFile)	**0**	1	**0**	1
adminFacet:may.invokeEnable(client,calcFacet)	**0**	1	**0**	1
adminFacet:may.invokeEnable(client,client)	**0**	1	**0**	1
adminFacet:may.invokeEnable(client,dFile)	**0**	**0**	**0**	**0**
adminFacet:may.invokeEnable(client,adminFacet)	**0**	1	**0**	1
adminFacet:may.invokeEnable(adminFacet,dFile)	1	**0**	1	**0**

on the passivity of `dFile` and `cFile`, there are other ways in which `dFile` can be influenced, possibilities that may not have been considered here.

As we mentioned earlier, it is crucial that we take all forms of authority propagation into account, when performing an analysis of eventual authority. Stack walking strategies unaided by such an analysis can only consider the propagation of influence between entities that are on the call stack at the same time. If `alice` can get a certain authority by calling `bob`, and consequently convey that authority to `carol` by calling `carol`, `bob` will effectively have used `alice` to propagate authority to `carol`, without `bob` and `carol` ever having called each other.

Part III

Related and Future Work

Chapter 9

Adding Authority Flow Constraints

We have shown in the previous chapters how we can calculate the minimal sets of restrictions in the configuration of permissions and in the behavior or subjects, that can guarantee the authority restrictions required by a certain safety policy. But safety policies can only be expressed directly in SCOLLAR as a set of predicates that should not be reachable (the safety goals). It would be useful if we could also express more elaborate policies directly.

In this chapter we provide a way to directly express elaborate policies as constraints on authority flow graphs, derived from the access graph. We concentrate on one particular flow graph constraint propagator, recently developed by Luis Quesada: the DomReachability [QVDC06] constraint propagator, who handles reachability constraints [QVD05] as well as path constraints.

This chapter is the result of joint work with the inventor of DomReachability: Luis Quesada, whose work is described in [Que06b].

9.1 Authority Flowing in Graphs

Directed graphs are useful for depicting binary relations directly, but we dismiss them for representing permissions and authority in general, because these relations can have arbitrary arity. For instance in SCOLL, the knowledge predicate `access` is binary, but the behavior predicate `sendTo` has three arguments and `did.getFromFor()`, a knowledge predicate, has four.

In capability systems, authority propagates from one subject to another. Therefore the propagation of authority can be presented in a labeled multigraph, in which the edge-labels represent the kind of authority that is flowing. Alternatively , we could consider a separate graph for every kind of authority whose propagation we want to graphically depict. The problem is then: how should we split up authority in "kinds-of-authority"?

If we make a coarse division into disjunct types of authority, we will get an over-approximation of the flow.

If we take the origin or the destination of the authority flow into account to decide the types, we get a very fine-grained division. For instance the "access-propagating-towards-`bob`" could be such an authority type. This kind of authority propagation is not directly visible in the access graph. It has to be derived with specific rules that account for the behavior of the subjects that are involved in the propagation. An edge will be added to the flow graph only when the collaboration between a set of entities results in the propagation of the authority along the arc.

A graph depicting the "access-flow-originating-from-`alice`" in a SCOLL pattern, will have an edge from `alice` to `bob` as soon as `bob` has received a capability from `alice`. If he can send that capability to `carol`, or if `carol` can get the capability from `bob`, there will also be an edge from `bob` to `carol`.

The resulting graph is a flow graph: a directed graph with a designated "source" node, in which the authority flow, represented by the arcs, propagates via the simple laws of transitive reachability. Many graph constraints can now be considered, which may be useful to express elaborate safety policies concerning the flow of that particular type of authority.

We investigate two kinds of such constraints in section 9.2 and show how they can be useful in this respect. Both constraints are propagated by the aforementioned DomReachability constraint propagator, that will be introduced in section 9.3. The use of reachability constraints in authority flow graphs is explained in section 9.4.

Remark

A subject can only use its own knowledge to decide its behavior and it will not be able, in general, to deduce the origin or the destination of its own access and information from that knowledge. Deriving the edges in an authority flow graph from the behavior of a set of subjects will usually over-approximate the actual causality of the propagation. If `bob` has received access-to-X from `alice` (X can be either data or a subject here) and he also sends this access-to-X to `carol`, that does not necessarily mean that `bob` is responsible for passing access-to-X from `alice` to `carol`. `bob` may have given access-to-X to `carol` regardless of the precondition: having received access-to-X from `carol`.

Whereas the use of flow graphs to depict authority flow can allow us to express elaborate safety policies directly as graph constraints on a flow-graph, the practical applicability of this approach may be limited due to:

- The approximation in the authority-propagating behavior of the subjects
- The approximation of authority propagation by plain transitivity in the derived graph

The more specific the authority expressed in the flow graph, the more precise we can expect these approximations to be.

9.2 Flow Graph Constraints

9.2.1 Definitions

Directed Graph

A directed graph or digraph G is a couple $\langle V, E\rangle$ where:

- V is a set of nodes (also called vertices).
  V is also denoted as: $Nodes(G)$

- E is a set of couples $\langle a, b\rangle$ called arcs (or directed edges), where a and b are nodes in V. An arc $\langle a, b\rangle$ is directed from a to b.
  E is also denoted as: $Edges(G)$

Subgraph of a Directed Graph

Let $G = \langle V, E\rangle$ and $G' = \langle V', E'\rangle$ be directed graphs.
G' is a subgraph of G, denoted $G' \subseteq G$
$\Leftrightarrow V' \subseteq V \wedge E' \subseteq E$

Finite Loopfree Paths in a Directed Graph

Let $G = \langle V, E\rangle$ be a directed graph, Let $a, b \in V$
The set of finite loopfree paths in G from a to b, denoted $Paths(G, a, b)$, is defined as the set of subgraphs P of G such that:

$$P \in Paths(G, a, b) \Leftrightarrow \begin{cases} Nodes(P) = \{k_1, \ldots, k_n\} : k_1 = a \wedge k_n = b \\ Edges(P) = \{\langle k_t, k_{t+1}\rangle \mid 1 \leq t < n\} \end{cases} \tag{9.1}$$

Note that for every node a in a graph, there is at least one path from a to a: the subgraph $\langle\{a\}, \Phi\rangle$.

Flow Graph

A Flow graph F is a couple $\langle G, s\rangle$ where:

- $G = \langle V, E\rangle$ is a directed graph
- $s \in V$ is a designated node called F's source.

Dominator Nodes

Given a flow graph $F = \langle G, s\rangle$, a node $a \in Nodes(G)$ is a dominator of another node $b \in Nodes(G)$ if all paths from s to b in G contain the node a. (Definition from: [LT79, SGL97]).

Formally : Let $F = \langle G, s\rangle$ be a flow graph and $a, b \in Nodes(G)$.

$$a \in Dominators(F, b) \Leftrightarrow a \neq b \wedge \forall P \in Paths(G, s, b) : a \in Nodes(P) \tag{9.2}$$

A node b is dominated by a node a in the flow graph F if $a \in Dominators(F, b)$.
A dominated node is a node that is dominated by at least one node.
Note that the nodes unreachable from s are dominated by all the other nodes.
Let us consider some other interesting properties on dominators:

Theorem 2. ***A reachable node cannot dominate any of its dominators***
If node a dominates node b and b is reachable, then b cannot dominate a.

Proof. From the definitions of $Paths$ and $Dominators$:
1) b is reachable

$$\Rightarrow \exists P \in Paths(G, s, b) : \begin{cases} Nodes(P) = \{s = k_1, \ldots, k_n = b\} \\ Edges(P) = \{\langle k_t, k_{t+1}\rangle\rangle \mid 1 \leq t < n\} \\ \forall i < n : k_i \neq b \end{cases}$$

2) $a \in Dominators(\langle G, s\rangle, b)$

$$\Rightarrow a \neq b \wedge \forall P \in Paths(G, s, b) : a \in Nodes(P)$$

$$\Rightarrow \forall P \in Paths(G, s, b) : \begin{cases} Nodes(P) = \{s = k_1, \ldots, k_n = b\} \\ Edges(P) = \{\langle k_t, k_{t+1} \rangle\rangle \mid 1 \leq t < n\} \\ \exists k_i = a : 1 < i < n \end{cases}$$

From 1) and 2) :

$$\exists P \in Paths(G, s, b) : \begin{cases} Nodes(P) = \{s = k_1, \ldots, k_n = b\} \\ Edges(P) = \{\langle k_t, k_{t+1} \rangle\rangle \mid 1 \leq t < n\} \\ \exists k_i = a : 1 < i < n \wedge \forall i < n : k_i \neq b \end{cases}$$

$$\Rightarrow b \notin Dominators(\langle G, s \rangle, a)$$ □

Theorem 3. *Of any two dominators of a reachable node c, in any loopfree path from the source to c, the first one in the path dominates the other one.*

Proof. Suppose that $a \neq b$, both dominate c and c is reachable. Then clearly, a and b are also reachable and there exists a path from the source to c that contains both a and b exactly once. Suppose a comes before b in any such path. Then that path contains a sub-path from b to c that does not include a.

If a would not dominate b then, since b is reachable, there would be a path from the source to b that does not include a, and one from b to c that would not include a either and we could use these to construct a finite loopfree path from the source to c that would not contain a, which is impossible since a is a dominator of c.

Thus a dominates b. □

Corollary 3. *The domination relation between the dominators of a reachable node is a complete strict order.*

Proof. A complete strict order is:

anti-reflexive : From the definition of Domination follows that no node dominates itself.

anti-symmetric : This follows directly from Theorem 2 and from the fact that all dominators of a reachable node are reachable.

transitive : Suppose a, b, and c, are dominators of a reachable node x and a dominates b and b dominates c. Theorem 3 says that the order of the dominators a, b, and c in any finite loopfree path from the source to x, decides which one dominates the other one. Since a dominates b, b will not dominate a (anti-symmetric) and thus a will come before b in that path. For the same reason, b will come before c in that path. This means that a will come before c in that path and from theorem 3 follows that a dominates c.

complete : Suppose a and b are dominators of a reachable node x. Then there is at least one path from the source to x and, like all other such paths, it contains both a and b. From theorem 3 follows that either a dominates b or b dominates a.

□

Immediate Dominator Nodes

Let $F = \langle G, s\rangle$ be a flow graph and let G be finite.
From corollary 3 follows that all reachable dominated nodes in F have a unique immediate dominator, which is defined as :

$$a = ImDominator(F, b) \Leftrightarrow \left\{ \begin{array}{l} b \text{ is reachable in } F \\ a \in Dominators(F, bj) \\ \neg\exists x \in Nodes(G) : a \in Dominators(F, x) \wedge x \in Dominators(F, b) \end{array} \right. \tag{9.3}$$

This allows us to represent the whole dominance relation as a tree, where the parent of a node is its immediate dominator. From now on we will consider only finite graphs.

Dominator Tree

Let $F = \langle G, s\rangle$ be a flow graph.
The dominator tree of F, denoted $DomTree(F)$ is the digraph $\langle V, E\rangle$ such that:

$$\left\{ \begin{array}{l} V = Nodes(G) \\ E = \{\langle a, b\rangle \in Edges(G) \mid a = ImDominator(F, b)\} \end{array} \right. \tag{9.4}$$

Extended Graph

The extended graph of a directed graph G, denoted $Ext(G)$ is obtained by replacing the edges by new nodes and connecting the new nodes accordingly as follows:

$$Ext(\langle V, E\rangle) = \langle V', E'\rangle : \left\{ \begin{array}{l} V' = V \cup E \\ E' = \{\langle a, e\rangle, \langle e, b\rangle \in V' \times V' \mid e = \langle a, b\rangle \in E\} \end{array} \right. \tag{9.5}$$

By extension of the definition, the extended graph of a flow graph $F = \langle G, s\rangle$ is defined as:

$$Ext(\langle G, s\rangle) = \langle Ext(G), s\rangle \tag{9.6}$$

Extended Dominator Tree

We call the dominator tree of a flow graph's extended graph its extended dominator tree:

$$ExtDomTree(F) = DomTree(Ext(F)) \tag{9.7}$$

Figures 9.1, 9.2 and 9.3 show an example of a flow graph, its extended graph, and its extended dominator tree, respectively. The extended dominator tree has two types of nodes: nodes corresponding to nodes in the original graph (node dominators) and nodes corresponding to edges in the original graph (edge dominators). The latter nodes are drawn in squares.

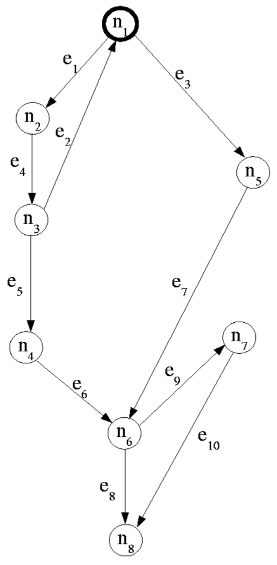


Figure 9.1: Flow graph

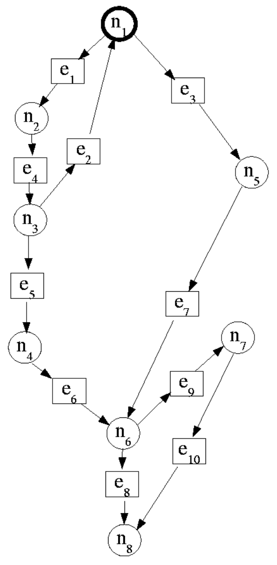


Figure 9.2: Extended flow graph

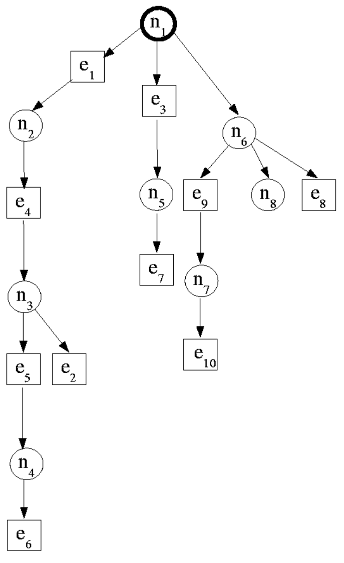


Figure 9.3: Extended dominator tree

9.3 The DomReachability Constraint

The DomReachability constraint is a constraint on three graphs:

$$DomReachability(F, D, T) \tag{9.8}$$

where

- $F = \langle G, s\rangle$ is a flow graph
- $D = ExtDomTree(F)$
- T is the transitive, reflexive closure of G, denoted $TC(G)$ and defined as

$$\begin{cases} Nodes(T) = Nodes(G) \\ Edges(T) = \{\langle a, b\rangle \mid Paths(G, a, b) \neq \Phi\} \end{cases} \tag{9.9}$$

From the definition of $Paths$ in (9.1): $\forall a \in Nodes(G) : \langle a, a\rangle \in Edges(T)$.

9.4 Constraints on the Reachability of Authority

9.4.1 The Bounded Transitive Closure Problem (BTC)

Given the directed graphs g_{min}, g_{max}, tcg_{min} and tcg_{max}, find a directed graph g such that:

$$\begin{array}{c} g_{min} \subseteq g \subseteq g_{max} \\ \text{and} \\ tcg_{min} \subseteq TC(g) \subseteq tcg_{max} \end{array} \tag{9.10}$$

The BTC instance can be directly modeled in terms of DomReachability: g_{min}, g_{max}, tcg_{min} and tcg_{max} are the bounds of fg and tcg.

Remark :

BTC problems are NP-complete. That was recently shown in [Que06a], by reducing the The Disjoint Paths Problem [GJ79] to BTC.

9.4.2 Safety and Liveness in terms of BTC

Basically, SCOLLAR deals with two concerns:

1. some authority should not be reachable for safety (safety properties)
2. some other authority should be reachable for functionality (liveness possibilities)

When the propagation of authority is safely approximated by the flow in a graph, both concerns can be expressed in the Bounded Transitive Closure Problem on this graph:

- the set of liveness possibilities will be tcg_{min},
- tcg_{max} will be the complement of the set of safety properties, and
- g_{min} and g_{max} will just be suitable bounds for the safe configuration of permissions we are looking for, decided by the initial configuration.

The next sections show examples of situations that allow us to put the reachability propagator to work, for simple problems expressed in SCOLL.

9.4.3 Confinement by Interposition

Suppose we have a set of previously unconnected, uncontrollable subjects and we want to find out how we can connect them, using controlled subjects, to allow them to perform their collaborative tasks, but also prevent them from breaking a given security policy. The tools we have to solve this problem are:

- a set of controllable subjects to be strategically inter-positioned between the uncontrolled subjects.
- a set of permissions to be granted to the controlled subjects.

The assignment is: find a configuration (graph) with a minimal number of controllable nodes (not exceeding a fixed practical upper limit), that guarantees the requirements for liveness (the uncontrolled subjects get enough authority) as well as the requirements for safety (the uncontrolled subjects do not get too much authority).

Practical Example

We take a well known example, expressing a simple Multi-Level Security Problem (MLS) [BL74]. Two external entities *Bond* and *Q*, with respective clearances *Top Secret* and *Confidential*, have to be given access to two external storage devices, one for *Top Secret* content and one for *Confidential* content.

We have to construct the content of a black box in (e.g. Figure 9.4), with a minimal number of subjects. Since the uncontrollable subjects cannot be restricted, their connection to the box is bi-directional. Even the devices are not trusted to be passive containers. They are unknown subjects and could be of any type.

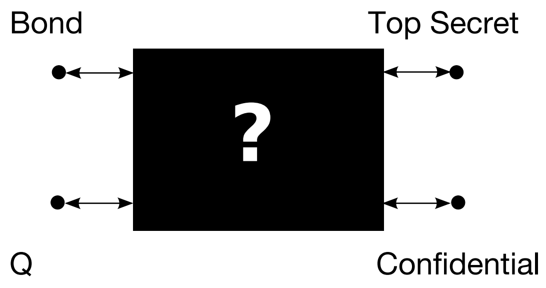


Figure 9.4: The $*$-property black box

The security relations we want to enforce between these four entities is: no top secret information leaks (down) to the confidential level. Therefore we will enforce the $*$-property (star-property) that states: agents should be able write to all levels above (and including) their own level of confidentiality and read from all levels below (and including) their own level of confidentiality, but no agent should be able to write strictly below his confidentiality level, or read strictly above his confidentiality level. This is a policy that specifies both liveness requirements and safety requirements, so we will express it as suggested in section 9.4.2.

Expressing the problem in terms of DomReachability

The BTC for the instance of the problem presented above is:

$$\begin{array}{rcl} g_{min} & = & \emptyset \\ g_{max} & = & \{\langle x,y\rangle | x,y \in \{b,q,t,c\} \cup \{o_1,o_2,...,o_{max}\}\} \\ tcg_{min} & = & \{\langle b,t\rangle, \langle t,b\rangle, \langle q,c\rangle, \langle c,q\rangle, \langle c,b\rangle, \langle q,t\rangle\} \\ tcg_{max} & = & g_{max} - \{\langle b,q\rangle\langle b,c\rangle\langle t,q\rangle\langle t,c\rangle\} \end{array} \tag{9.11}$$

In the problem b stands for Bond, q for Q, t for the top-secret device, and c for the confidential device. The controlled nodes are o_1, o_2,...,o_{max}.

Apart from the BTC constraints, we have to express the fact that b, q, t, and c are uncontrolled, by making sure that all their connections are bi-directional. We therefore added the necessary implication constraints to the problem:

$$\forall 0 \le i \le max, x \in \{b,q,t,c\} : \langle x,o_i\rangle \in g \Leftrightarrow \langle o_i,x\rangle \in g \tag{9.12}$$

To minimize the number of controlled subjects, we can start with zero controlled nodes and iteratively add one more, until we find a solution.

A solution for the authority flow with a minimal number of internal nodes is presented in Figure 9.5.

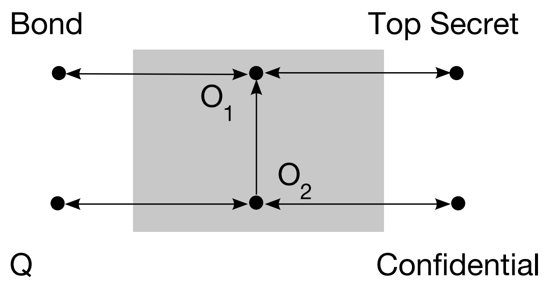


Figure 9.5: A solution with the minimal number of controlled subjects

This solution is still in the form of an authority-flow graph and has to be transformed to a SCOLL configuration. Let us do that for capability systems, keeping in mind that the connections to/from the four unrestricted subjects are outside the system and they cannot be controlled: they are assumed to be initially separated and our set-up is only responsible for not jeopardizing the original confinement.

This process is straight forward, but not unique. We could for instance map all arcs, including the one between O_1 and O_2 to both-way access permissions in the SCOLL configuration. O_1's behavior must be restricted so that it does not send information to O_2. O_1 can be given initial knowledge about who is *Top Secret* and his behavior should simply accept incoming data as responder only and forward that data as an invoker to *Top Secret*. In that case, O_2's behavior must be restricted so that it does not get information from O_1.

In fact, now the structure of the graph has been generated by DomReachability, SCOLLAR can also be used to calculate the necessary behavior restrictions for $O1$ and $O2$.

9.4.4 Confinement by Restricted Behavior

In the previous section we relied on the internal subjects to behave exactly as allowed. This is typical for capability systems [DH65]: such relied-upon subjects are called capabilities. Notice however, that the behavior restrictions were static and binary and could therefore be expressed directly in a graph. Such configurations can, in principle, also be protected with a reference monitor that checks the read/write permissions between the internal subjects, before they are exerted.

Of course, behavior can be restricted in much smarter ways than simply emulating the inhibition of certain permissions. We can try to program capabilities to use their permissions (access) in a way that makes the desired authority reachable while preventing all illegal authority. Taking such smart behavior into account allows for a more accurate analysis of the reachable authority in a system. An account of the different levels of detail in which the boundaries of authority can be calculated is given in [Mil06b].

Suppose we want to express the behavior of a subject that only passes information if:

- other subjects wrote that information to it (it did not read the information itself from other subjects), and
- it writes that information itself to other subjects (it does not reveal that information to its own readers)

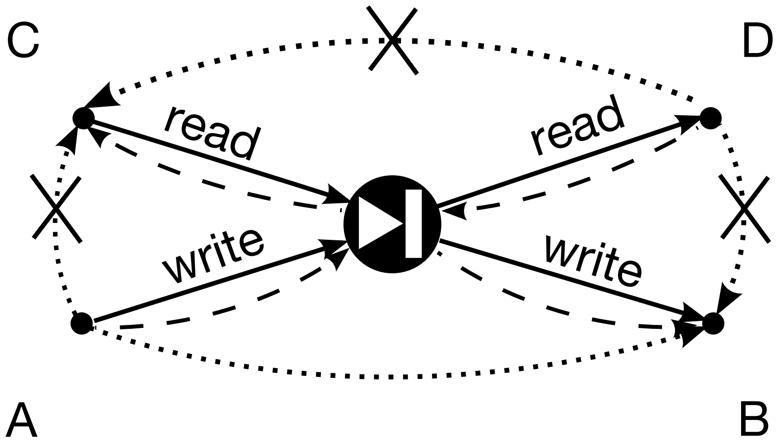


Figure 9.6: Data Forwarder (dataflow diode)

Such a subject acts as a forward diode for data flow, depicted in figure 9.6. The full arcs denote the access rights and the dashed arcs represent the corresponding flow of data. The data-flow is only transitive in one direction: from A to B, as indicated by the dotted arcs. The behavior of the diode in the middle prevents data to pass in the three other directions.

Expressing this behavior in SCOLL is straight forward, but let us now try to express it as a flow graph. Instead of being a simple node, the subject's behavior will now be presented by a subgraph of the complete flow graph. This subgraph has four nodes: two $in-ports$ and two $out-ports$, one of each kind for reading and the other one for writing. All external arcs will be connected to one of the four ports: the incoming flow to the in-ports, the outgoing flow to the out-ports, the flow via read permissions to the read-ports, and the flow via write permissions to the write-ports. These restrictions can directly be expressed in BTC, by removing the illegal external connections from g_{max}.

Table 9.1: Subgraphs for behavior-based internal dataflow

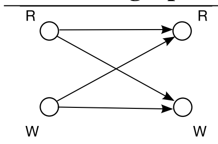
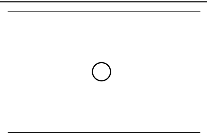
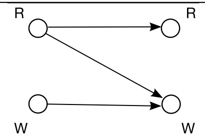
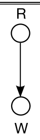
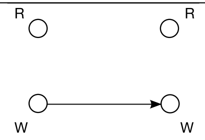
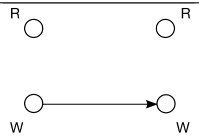
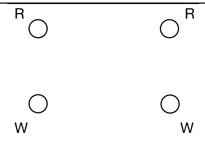
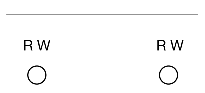
behavior graph	**simplified graph**	**behavior**
		unrestricted behavior
		hides its writers data from its readers
		data forwarder of figure 9.6
		non-transparent subject

Table 9.1 shows some behavior subgraphs with four internal ports (not to be confused with the graph in figure 9.6). The internal flow (arcs) always goes from an in-ports (left) to an out-ports (right). These subgraphs are to replace the monolithic subject nodes in the data-flow graph. The arcs here correspond to the dotted arcs (flow-through) in the example of figure 9.6. Depending on which of the four possible arcs are present, the behavior-graph can be simplified (second column of table 9.1).

This suggest the utility of graph-based behavior definitions in patterns of capability based collaboration, to improve the accuracy of the approximation of authority analysis in flow graphs. Compared to SCOLL's expressive power, the behavior that is expressed with these 4-node subgraphs is very simple. Adding extra expressive power will require a fast growing number of nodes in the subgraph. Still, the example shows that the approach is useful beyond modeling simple all-or-nothing use of permissions.

Now that the behavior is part of the flow graph, the solution to a safety problem can suggest restrictions to the behavior of trusted subjects as well as restrictions to the initial configuration of permissions, just like in SCOLLAR.

9.4.5 Implication graphs: The Conditional BTC Problem

In the previous sections we had to use additional constraints to express the security problems. For instance, we used extra constraints for all four uncontrolled subjects in section 9.4.3, to express that they should take only bidirectional connections. The BTC problem can easily be extended to incorporate such implication constraints.

A condition like: "if an edge $\langle A, B\rangle$ is in the graph, then so should $\langle B, A\rangle$" can be seen as edge between edges: $\langle A, B\rangle \rightarrow \langle B, A\rangle$, in a graph whose nodes are edges in the original graph and whose edges represent implications.

But the approach doesn't need to be limited to conditions between edges in the same graph: we could as well express inter-graph conditions as edges. Such a condition can be presented as an edge from an edge-in-one-graph to an edge-in-another-graph. The edges we connect by implication can be chosen from any graph in the BTC problem. To further improve the expressive power, we also allow edges to be chosen from the complement of such a graph. We denote the complement of graph $g = \langle V, E\rangle$ as $g' = \langle V, (V \times V) \setminus E\rangle$.

Definition

Given the directed graphs g_{min}, g_{max}, tcg_{min} and tcg_{max}, and given a graph $condg$ that is a subgraph of $\langle V, E\rangle$ such that :

$V = Nodes(g_{max}) \uplus Nodes(tcg_{max}) \uplus Nodes((g_{max})') \uplus Nodes((tcg_{max})')$,
and

$$E = V \times V = \{\langle e_1^{G_1}, e_2^{G_2}\rangle \quad | \quad G_1, G_2 \in \{g_{max}, tcg_{max}, (g_{max})', (tcg_{max})'\} \\ \wedge e_1 \in Edges(G_1) \wedge e_2 \in Edges(G_2)\}$$

find a directed graph g such that:

$$\begin{gathered} g_{min} \subseteq g \subseteq g_{max} \\ \text{and} \\ tcg_{min} \subseteq TC(g) \subseteq tcg_{max} \\ \text{and} \\ \forall \langle e_1^{G_1}, e_2^{G_2}\rangle \in condg : e_1 \in G_1 \Rightarrow e_2 \in G_2 \end{gathered} \tag{9.13}$$

By constructing implication-edges with edges from the complement of a graph, we can express negative requirements too.

The security problems in sections 9.4.3 and 9.4.4 are direct applications of CondBTC. The implications involving edges of the solution graph and its transitive closure can be directly represented in terms of $condg$.

Expressing the problem in terms of CondBTC

The CondBTC for the $*$-property exercise of section 9.4.3 is:

$$\begin{aligned} g_{min} &= \emptyset \\ g_{max} &= \{\langle x, y\rangle | x, y \in \{b, q, t, c\} \cup \{o_1, o_2, ..., o_{max}\}\} \\ tcg_{min} &= \{\langle b, t\rangle, \langle t, b\rangle, \langle q, c\rangle, \langle c, q\rangle, \langle c, b\rangle, \langle q, t\rangle\} \\ tcg_{max} &= g_{max} - \{\langle b, q\rangle\langle b, c\rangle\langle t, q\rangle\langle t, c\rangle\} \\ condg &= \{\langle x, y\rangle, \langle y, x\rangle \mid x \in \{b, q, t, c\} \wedge y \in \{o_1, \ldots, o_{max}\}\} \end{aligned} \tag{9.14}$$

9.5 Future Work

9.5.1 Implication hypergraphs: The Cardinal BTC Problem

We observe that the expressive power of CondBTC can be further enhanced if we use the $condg$ graphs to represent, instead of implications between edges in $g \uplus g' \uplus TC(g) \uplus (TC(g))'$, implications between mixed sets of such edges. An edge $\langle A, B\rangle$ in $condg$ can then represent a composite condition: if all edges in the set A are present, then so should at least one edge in the set B. We call this the Cardinal BTC Problem(CardBTC).

The extended definition of $condg$ allows us to simplify the definition of the problem. The BTC graphs g_{min}, g_{max}, tcg_{min} and tcg_{max} themselves can now be defined with $condg$, transitive closure, and graph complement as follows:

$$\begin{array}{rcl} \forall e \in g_{min} & : & \langle \emptyset, \{e^{g}\}\rangle \in condg \\ \forall e \notin g_{max} & : & \langle \emptyset, \{e^{(g)'}\}\rangle \in condg \\ \forall e \in tcg_{min} & : & \langle \emptyset, \{e^{TC(g)}\}\rangle \in condg \\ \forall e \notin tcg_{max} & : & \langle \emptyset, \{e^{(TC(g))'}\}\rangle \in condg \end{array} \tag{9.15}$$

The expressive power of CardBTC can be extended even further, if we also label the edges with constraints on the cardinality of the target set.

Implementation Possibilities

The cardinality constraints involved in CardBTC can be imposed by standard approaches based on cardinality propagators [VD91].

Observe that, when we associate each basic graph constraint with a literal, a Boolean Satisfiability Problem appears. This abstraction could allow us to take advantage of BDD propagators to narrow down the literals composing a given disjunction [HLS05]. Hybrid approaches, like the one suggested in [HS06], can also be considered, to exploit the advantages of SAT solvers.

Applying CardBTC for practical Security Problems

CardBTC allows us to express complex conditions on the propagation of authority in several ways we did not yet explore:

- It can be used to express more complex ways of authority propagation than transitive closure.
- It can be used to represent fine-grained conditional behavior of trusted subjects, without the need to represent every subject as a complex subgraph.

9.5.2 Towards a Synergy of SCOLLAR and DomReachability

We expect the applications of DomReachability for security to be most useful in collaboration with our existing SCOLLAR tool. SCOLLAR is most suitable to express a system's rules that govern the propagation of permissions and authority, and a subject's behavior. System rules can express realistic models for propagation, that can take the restrictions in the behavior of the trusted subjects into account. Subject behavior can be expressed in a way that depends on the information that a subject has from initial conditions and has acquired during the collaboration with other subjects. Its expressive power makes SCOLLAR a tool that can (also) be used to study the propagation of authority in capability systems and patterns of collaborating entities.

The restriction to monotonic approximations (that are safe but may possibly be too crude) prevents us to directly express the revocation of authority. This is relevant for capability systems too because, even if access permissions cannot be revoked, it is very well possible (and easy) for a subject to revoke the authority it used to provide to its clients, for instance by refusing to collaborate any further and no longer pass on any data or capabilities.

This is where the dominator part of DomReachability can be of direct use. Instead of simply stating that some effect (authority) should be prevented, we could instead require that all authority of a certain kind should only ever be available via a trusted subject that is able to revoke the authority by (further) restricting its collaborative behavior. In the authority-flow graph, to be derived from the access-graph, a trusted subject `alice` can revoke all `bob`'s authority over a third subject `carol`, if `alice` dominates `bob` in the authority-flow graph that originates with `carol`, or if `alice` has the authority to instruct another dominator (e.g. `caretaker` in section 8.2) to do so .

Chapter 10

Designing a Capability Secure Language

10.1 Introduction

10.1.1 Motivation

Most of the work presented in this thesis was motivated by an indirect goal: the transformation of the multi-paradigm language Oz into a capability secure language, that enables and facilitates the practice of secure programming. That project is called Oz-E, in honor of the capability secure programming language E [MSC$^+$01], who embraces a pure object-oriented paradigm, but also provides support for declarative and functional programming.

10.1.2 Revisited Concepts

To define secure programming, we will quickly revisit some important concepts that were first explained in section 1.3.

Definition 32 (Permission)**.** *An action an entity (subject) is allowed to perform. No subject can perform an action without having the permission. The subject having the permission may or may not be able to perform the action. If it is able to perform the action, it may or may not actually do so. Performing the action may or may not have an effect. If it has, the nature of the effect may not necessarily be deterministically defined by the action alone.*

Permissions are usually binary relations between subjects: they allow the holder subject to perform a certain action on the target subject.

Definition 33 (Capability)**.** *An unforgeable reference that inextricably combines a permission with the designation of the target subject and with the ability to use the permission. A capability can simply be a reference to a target subject, if such a reference cannot be forged. In that case, the permission it carries is: to use the designated target (e.g. invoke it, or use it as an argument in other permitted invocations).*

Definition 34 (Authority)**.** *An effect a subject is able to invoke. The subject having a certain authority may or may not actually use it. Authority to invoke an effect implies the permission to perform an action that will (eventually) cause that effect. The effect* *of using a permission is not always determined by the permission and its holder alone. Permissions represent no more than a crude approximation of a subject's authority, as it can be influenced by the behavior of the holder and the target subject.*

Definition 35 (Secure Programming)**.** *Programming for interaction with components of unknown or uncertain reliability, while guaranteeing that a predefined level of vulnerability is not exceeded.*

Secure programming has to guarantee two conditions:

1. All relied-upon components are programmed reliably so that they
    - do not actively abuse their authority to inflict unacceptable damage and
    - cannot be lured into doing so by their collaborators (see section 8.1: confused deputies).
2. No authority to inflict unacceptable damage can become available to any component unless it is relied-upon not to inflict such damage (see section 8.3: confinement).

Given a program and a non-trivial set of safety requirements, the programmer can model his/her program in SCOLL (Chapter 6) and analyze its vulnerabilities in SCOLLAR (Chapter 7) to acquire the necessary confidence that the program meets its safety requirements. If it is not, SCOLLAR can compute alternative ways to design the program such that safety is guaranteed (Section 7.2.2). If SCOLLAR cannot find such alternatives, this means that:

- the SCOLL model should be refined to better approximate the program's authority propagation semantics, or
- the program is inherently unsafe: its functionality requirements violate its safety requirements, regardless of its implementation.

In general, there is no way to differentiate between both causes of failure. However, insight into the mechanisms that cause violation of safety will be gained from the analysis and can provide useful hints towards alternative designs. Patterns of collaboration with untrusted entities, such as provided in chapter 8, represent a crystallized form of experience in secure programming and can provide ready-made solutions to common problems.

The programming language has a crucial influence on the maximum propagation of authority among interacting entities, in programs that are expressed in the language. This influence is modelled in the system rules of SCOLL (Sections 5.3.3 and 6.2.2).

Definition 36 (Behavior)**.** *Behavior is a safe (over) approximation of an entity's own use of permissions and of its influence on the authority that is provided by permissions of which the subject is the target. Behavior is the subject's own positive influence on authority, we could not with certainty rule out from the partial knowledge we have about it.*

Definition 37 (Collaboration)**.** *Collaboration is the form of interaction between two (or more) entities, allowed by a permission, whereby the behavior of the target subject can completely annihilate the effect of the interaction (except for the fact that the interaction itself took place). For a formal definition in terms of SCOLL's system rules, see definition 29 in section 6.8.7.*

The design principles for Oz-E will be partly dictated by the requirement for collaboration: subjects should have the power to restrict authority and its propagation, whether as a holder or as a target of a permission.

10.1.3 Approach

We gained insight in the most important principles for secure language design from two main sources:

- Earlier experiences with the capability-secure language E [MSC$^+$01] and the W7-kernel for Scheme 48 [Ree96].
- Experience with SCOLLAR (see chapter 8).

Part of the content in this chapter was published earlier as [SV05]. We do not claim that the set of guidelines we propose here is complete, but we are confident that they represent a useful and valid contribution to secure language design.

While the Oz language was designed to satisfy strong properties, such as:

- full compositionality,
- lexical scoping,
- simple formal semantics, and
- network transparent distribution.

Security, in the sense of protection against malicious agents, was not a design goal.

Following Mark Miller's suggestion in [Mil03], we do not try to add security to Oz, but instead to remove its insecurity. Therefor Oz-E will start off as a small subset of Oz that is known to be secure. It will gradually grow in functionality and expressive power, while keeping the language secure.

The ultimate goal is to reach a language that is at least as expressive as Oz, but is secure both as a language and in terms of its implementation. Most important: it should be straightforward to write programs in Oz-E that are secure against many realistic threat models.

10.2 Basic Principles

We distinguish between three kinds of principles:

- Mandatory Principles: to make secure programming possible
- Pragmatic Principles: to make secure programming feasible
- Additional Principles: to support analysis of authority propagation

All principles serve a common goal: to support the development of programs that use untrusted modules and entities to provide (part of) their functionality, while minimizing their vulnerability to incorrectness and malicious intents of these entities.

To avoid excess authority, secure programming has to apply the Principle of Least Authority (POLA) (Section 1.3.4) with scrutiny. That means that the language should support POLA in two ways:

- Make it easy for the programmer to fine-grain and minimize the authority that is directly provided to entities, both relied-upon and untrusted.
- Make it hard or even impossible for the programmer to build entities whose authority can be abused by adversaries or incorrectly programmed allies.

The confused deputy attack (Sections 4.6 and 8.1) is an important form of abuse, the language should help the programmer to prevent. In short, a deputy is an entity that is designed to get authority from its clients and to use that authority to perform a task on that client's behalf. The deputy is confused when he uses his own authority instead of the client's. The deputy can prevent confusion if he requires his clients to delegate authority in the form of a capability.

We therefore limit our attention to languages that support capabilities.

10.2.1 Mandatory Principles

An simple and effective way to build a capability based language is: to make every reference (pointer) unforgeable in the language. That way, every reference becomes a capability, designating the referenced entity and carrying the full permission to "use" or "access" the entity.

When all authority is accessible only through such unforgeable references, no other permissions are necessary to build a capability secure language. Instead of building a capability that only allows partial use of its designation, we can make a full-permission capability that designates an proxy for the original entity and construct the proxy so that we can rely on its restricted behavior to use only the intended part of its authority.

For instance, instead of creating a read-only capability designating a file, create a full-access capability to a readstream on that file.

Three principles express what is necessary and strictly sufficient to enable capability based secure programming:

1. Give entities no authority that is not rooted in capabilities
2. Upon creating or loading an entity, provide it with no capabilities.
3. Limit the possibility for entities to get extra authority from combining two capabilities.

No Ambient Authority

This is the most strict requirement for a language that is to allow secure programming. It combines the first two principles mentioned above.

Ambient authority is authority that is available to an entity "from the environment". If entities can get authority this way, the programmer can only restrict the use of that authority in the part of the program that he writes himself.

To enable the programmed entities to have complete control over the propagation of authority, the language has to cut off every other way of getting authority.

Therefore, all entities should come to live with no default authority and capabilities should only be acquired in the following ways:

1. By endowment and parenthood (as defined in section 4.3.3). These are forms of authority propagation between a creating entity (parent) and the entity it created (child). By parenthood, the parent gets the capability to use the child. The parent

is allowed to endow the child with a subset of his capabilities. These are the only two forms of authority propagation that do not require collaboration.

2. By collaboration. Using a capability, an entity can initiate a collaboration with the entity designated by the capability. (See definition 37).

The language thus has to make sure that no authority can be acquired in any other way, for instance via globally or dynamically scoped variables or via memory probing and forging.

Concrete, this means :

- The language should be purely lexically scoped.
- The language and the implementation should be completely memory safe.
- The loader should provide no authority to entities upon loading them into memory.

Avoid Authority Amplification

This principle is a variant on the first one.

In some situations, the availability of two capabilities can result in more authority than the simple sum of both authorities. This phenomenon is called authority amplification. It has useful applications for authentication purposes, which will be described in section 10.2.2.

However, authority amplification should not be used as a means to distribute authority, as will be shown in the following example.

Example:

> *Consider a file system in which the file references are capabilities that carry minimal permissions. If you have such a capability, you can use it to test if the designated file exists, but not create the file, or read from it or write to it. To read or write the file, you also need access to the module "FileIO", which is only available to some trusted entities. That module provides read and write operations in the form of unforgeable procedure references that take an input argument (a file reference) and then read or write from it on its client's behalf.*
>
> *On its own, access to such an operation does not provide any authority, but combined with a simple file reference, it amplifies the authority of that reference. The file reference, even if unforgeable, is used here as a designation, separated from its actual authority. Therefore it makes the operations vulnerable to confused deputy attacks.*

Authority amplification causes extra "ambient authority" to become available to an entity and can confuse deputies because the extra authority is neither provided by the client nor by the deputy.

Figure 10.1 shows what can happen in general. In the lower plane we see the client and the deputy: the client passes a capability to the deputy. In the middle plane, the ovals represent the capabilities held by the client and by the deputy. In the upper plane, the ovals represent the authority reachable via the client's capabilities and via the deputy's capabilities.

The passed capability designates an entity. Authority amplification will increase the authority that the deputy has over this entity. This is shown by the small bold

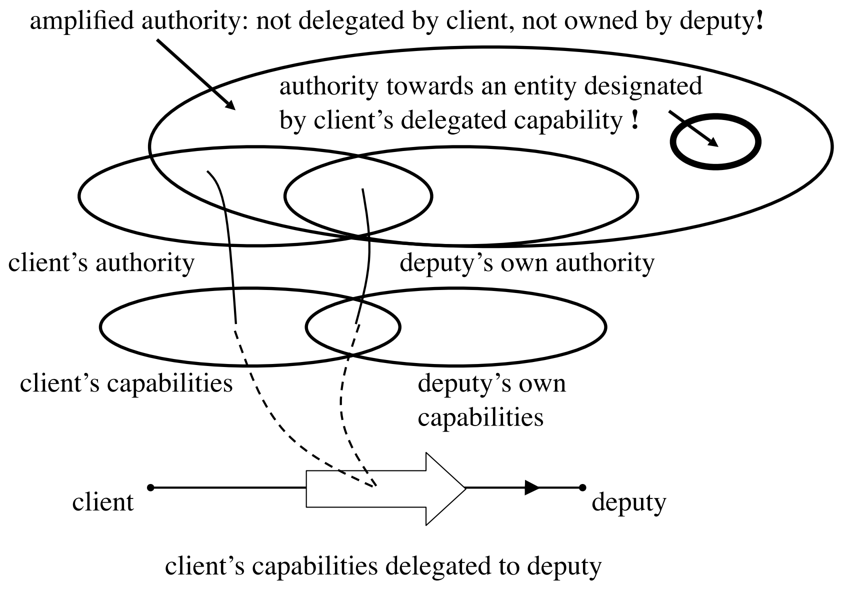

amplified authority: not delegated by client, not owned by deputy!
authority towards an entity designated by client's delegated capability !
client's authority
deputy's own authority
client's capabilities
deputy's own capabilities
client
deputy
client's capabilities delegated to deputy

Figure 10.1: Authority amplification can confuse deputies

oval. In a way, designation and authority have effectively become separated again. The confused deputy problem arises.

In this respect, it is relevant to consider the differences between types of data abstractions made in [VH04]. Abstract data types (ADT's) are called unbundled because they provide operations separately from values, while bundled data types combine the data with the operations. As shown in the example, unbundled data types will lead inevitably to confusion of deputies via authority amplification. Oz-E will therefore, contrary to Oz, opt for bundled data types.

Further on, we will take this reasoning further and plead for object-style data types to maximally support authority control via collaboration.

10.2.2 Pragmatic Principles

Due to the mandatory principles, all essential control of authority distribution and propagation becomes available to programmers. They can now, in principle, start building entities that will perform reliably in collaboration with untrusted ones.

However, it is not enough for a language to enable secure programming. It should also make secure programming feasible in practice and consequently favour secure programming as the default. The principles in this section are meant to promote and facilitate secure programming.

Defensive Consistency

The dominant pattern of secure programming, which the language must make practical, is that clients may rely on the correctness of servers, but that servers should not rely on the correctness of clients.

A server (in general, any “callee”) should always check its preconditions. A client (any “caller” in general) may rely on the server, if it has the means to authenticate the server. The usefulness of this pattern has emerged from experience with E and its predecessors. Miller puts this requirement in a larger context with other forms of robustness [Mil06b]. In section 4.1.6 we saw that Dennis and Van Horn identified a very similar concern [DH65].

In traditional correctness arguments, each entity gets to rely on all the other entities in the program. If any are incorrect, all bets are off. It may be OK to rely on untrusted entities for the correct functionality of a program, but there is no reason to also rely on their security. On the other hand, for programmers to actually check all preconditions, postconditions, and invariants is not a realistic approach. *Defensive consistency* is when every entity explicitly checks its input arguments when invoked. This is a realistic and effective middle way.

A secure programming language must make it practical to write most abstractions painlessly to this standard. Its libraries should be populated by abstractions that live up to this standard. The remaining abstractions should explicitly state that they fall short of this standard.

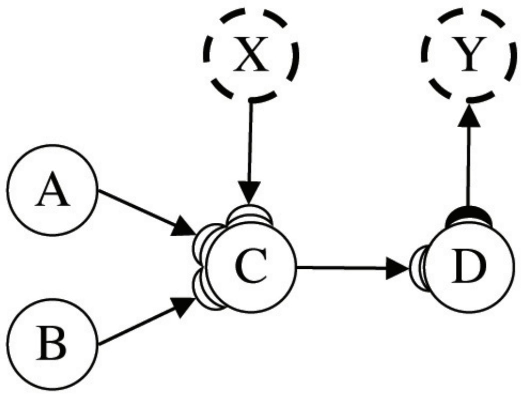


Figure 10.2: Paths of vulnerability

Figure 10.2 shows an access graph. Dashed nodes are entities not relied upon in any way. White crescents indicate that all incoming arguments are checked before being accepted or used. A black crescent indicates that all outgoing arguments are checked before being transferred. `A` and `B` are vulnerable to (rely upon) `C` and `C` is vulnerable to `D`. Since vulnerability is a transitive relation, `A` and `B` are also vulnerable to `D`. Because `C` checks its incoming arguments when invoked, it will protect itself and its clients from malicious arguments (e.g. provided by `X`). Paths of vulnerability are easy to follow and go one way only. Two clients vulnerable to the same server are not for that reason vulnerable to each other.

To support defensive consistency, Oz-E has to make it easy for the programmer to check incoming arguments. Guards, authentication primitives, and auditors, presented in the next sections, can provide such support.

Guards

E's guards [Sti00, Mil06b] form a soft typing system [CF91] that provides syntax support to make dynamic checking as easy as using static types. Guards are first class citizens and support arbitrary complex dynamic checking without cluttering the code with the actual tests. They can be user defined, and combined into more complex guards by logical operators.

Authentication

For an entity to defend its invariants in a mutually distrusting context, it can be important to know the origin of collaborating entities. The entity may want to authenticate a procedure before invoking it, and an argument before applying the procedure to it or before returning a capability to its invoker. Because capabilities unify designation and permission, and because the confused deputy problem can be naturally avoided, there is no need to identify the invoker.

We do not necessary want to know who wrote the code for that entity – since that knowledge is not very useful in general – but whether we can rely upon the entity that loaded it and endowed it with initial authority. For example, if we rely upon bank B, we can authenticate an account-entity A by asking B if A is genuine, in other words if B recognizes A as one of the accounts B – or maybe an associated branch – created earlier.

There is a tension between the need for authentication and the requirement that all interaction should happen via collaboration. Collaborating with A to find its identity would give A the possibility to lie about it. On the other hand, allowing to break A's encapsulation would deny A the possibility to control authority propagation. The compromise here is to rely on A's alleged creator B, who, if he really created A, has had rightful access to A's identity and internal data, at the time of creation. B does not have to collaborate with A to find out if it is really one of its creations.

Authentication by Invited Auditors

The above form of authentication is only useful to authenticate entities of which the alleged creator is a relied-upon third party. Moreover, this form of authentication cannot tell us anything further about the actual state of an entity at the time of authentication.

To reliably interact with entities of unknown origin, it must be possible to have them inspected by a relied-upon third party. Without breaking encapsulation, that can be done as shown by E's auditors [YM00, Mil06b].

When an entity is created, a relied-upon third party auditor is invited by the creator, to inspect the entity's behavior and lexical scope. Later, when the auditor is asked to vouch for the relied-upon properties, it will reveal its conclusions, or if necessary re-inspect the state of the entity before answering yes or no. If inconclusive or uninvited, it will answer no.

Failing Safely

When an entity cannot guarantee its invariants in a certain condition, it should raise an exception. The default mechanism should not enclose any capabilities or potentially sensitive information with the exception that is raised. Part of this concern can be automated by the guards discussed earlier, who will throw an exception on behalf of the entity.

Preemptive Concurrency and Shared State

Preemptive concurrency enables an activation of an entity at some point in its progress to destroy the assumptions of another activation of the same entity at another point in its progress. This phenomenon is called plan interference.

Semaphores and locks give programmers control over the interaction between concurrently invoked behavior, but their use is error-prone and increases the overall complexity of a program. Good locking becomes a balancing exercise between the danger of race conditions and deadlocks. Preemptive concurrency with shared state makes defensive programming too hard because considering a single invocation of behavior is not enough to ensure preconditions and invariants.

For example, consider a simple observer' pattern [GHJV94]. With message-passing concurrency as explained in chapter 5 of [VH04] – all entities involved are *Active Objects*, subscription is done by providing a *Port*, and notification via a *Port.send* operation – all update notifications of an entity are guaranteed to arrive at the subscribers in the order of the update.

10.2.3 Additional Principles: Support for the Review Process

When the language is ready to provide all the necessary support for secure programming, one more important design concern remains. The programmers are now in the position to avoid security flaws while programming, but they also need to be able to quickly find any remaining vulnerabilities that might have got in.

Oz-E must be designed to make security debugging easy. Its syntax should therefore allow programmers to quickly identify big parts in a program that are obviously safe, and concentrate on the remaining part.

A minimum set of tools to support debugging and analyzing the vulnerabilities is indispensable. These can range from syntax coloring to debuggers for distributed code and tools for security analysis.

The latter is the main rationale for the other chapters in this thesis, leading up to the SCOLLAR tool described in section 7. SCOLLAR is meant primarily to allow us to examine the usability of patterns of safe collaboration that emerged from experience (e.g. the Powerbox [SM02] and the Caretaker[MS03, Red74]), and enable the discovery of new such patterns.

10.3 The Layered Structure of Oz-E

The Oz language has a three-layered design. We briefly introduce these layers here. For a detailed explanation, see [VH04] (chapter 2 and appendix D).

The lowest layer is a simple language, kernel Oz, that contains all the concepts of Oz in explicit form. The next layer, full Oz, adds linguistic abstractions to make the language practical for programmers. A linguistic abstraction is an abstraction with syntactic support. In general, an abstraction is a way of organizing a data structure or a control flow such that the user is given a higher-level view and does not have to be concerned with its implementation. The final layer, Mozart/Oz, adds libraries and their interfaces to the external environment that depend on the functionality provided by the operating system.

We realize that in an ideal world, the language and the operating system should be developed together. Pragmatically, we will provide as much of the operating system functionality as possible inside the third layer of OzE. Any remaining functionality – not fitting the language without a complete rewrite of the operating system – will be accessible through a general system interface.

The importance of the layered architecture for security is stressed by a flaw in the current Mozart system that was found by Mark Miller. The module *Time*, currently available in the second layer as ambient authority, provides access to the system clock and should therefore be transferred to layer three, the functionality of which can only be available via explicitly granted capabilities.

Read access to the system time can be used to create indirect channels for data-communication that are invulnerable to countermeasures like randomness in thread execution sequence and adding randomized delays. While such countermeasures cannot prevent an entity to broadcast data using covert channels, they can make it arbitrary hard for them to receive their instructions and input via such channels.

Oz-E should keep as much of this layered structure as possible, while staying within the boundaries of the security requirements. We will start with very simple versions of these layers and grow them carefully into a full-featured language, maintaining the security properties throughout the process. The project will start by showing formally that the initial versions of kernel language and full language are secure. During the growth process, a formal semantics of the kernel language should be maintained at all times, to allow verification of these principles.

In the following three subsections, we present each of the three layers and we discuss some of the issues that need to be resolved for each layer. Again, we do not claim to be complete in this respect.

10.3.1 Kernel Language

The kernel language should be complete enough so that the programmer never needs to go below that level, e.g., to a byte code level. Making the kernel language the lowest level seen by (normal) application developers and library designers, reasoning and program development and analysis will be simplified. Only the language designers themselves will go below that level. The implementation will guarantee that the kernel language satisfies its semantics despite malicious interference by programs written in it.

The initial kernel language will be as close as possible to the general kernel language of Oz, which has a complete and simple formal semantics as given in chapter 13 of [VH04]. This is the most complete formal semantics of Oz that exists currently. As far as we know, the relevant part of the Mozart system implements this semantics. It is straightforward to show that this kernel language satisfies basic security properties such as secure closures (encapsulation based on lexical scoping), absence of ambient authority, and unforgeable identity of kernel language entities.

In the rest of this subsection, we address two specific issues that are directly related to the kernel language, namely authentication and finalization. Authentication is an issue that is directly related to security. Finalization is an issue that is indirectly related to security: the current design has problems that would make building secure systems difficult.

We prefer the kernel language of Oz-E to be a subset of the full Oz-E language. This will result in semantic clarity, uniformity of syntax, and simplicity, all important pedagogical assets when teaching or learning Oz-E. Furthermore, the kernel language subset will allow us to experiment with language extensions while staying within the language.

Authentication via Token Equality

A basic requirement for building secure systems is authentication of authority-carrying entities. Entities that were created by relied-upon third parties should be recognizable with the help of the third party. This means that the entity needs an identity that is unforgeable and unspoofable, otherwise a creator could never be sure the entity is really the one it created earlier. Unforgeable means that it is impossible to create an identity out of thin air that matches with the identity of an existing entity. Unspoofable means that the authenticity check cannot be relayed (man in the middle attack).

The kernel language has to let us achieve these properties for its own authority-carrying entities and also for user-defined entities built using the kernel language. Both of these categories impose conditions on the kernel language semantics.

In this section, we examine these conditions. In the following paragraphs we use the term “entity” to mean a language entity of a type that can carry authority (be a capability), as opposed to pure data. Data has no identity. Oz has an equality operator “==” that implements structural equality between data.

For kernel entities, authentication is achieved by the kernel language syntax and semantics. The kernel semantics ensures that each newly created entity has a new identity that does not exist elsewhere and that is unforgeable.

For user-defined entities, authentication has to be programmed. For example, a user-defined entity type called “object” could use one-argument procedures for its implementation. The identity of an object should then not be confused with the identity of its procedure.

This implies that the kernel language should have operations to build unforgeable and unspoofable identity into user-defined entities. The concepts *chunk* and *name* from the Oz kernel language can be used for this purpose. A *name* is an unforgeable constant with only one operation: token equality. A *chunk* is a record with only one operation: field selection. Its field names are hidden and can be *names* (unguessable).

With chunks and names, it is possible to build an operation that wraps an entity in a secure way, so that only the corresponding unwrap operation can extract the entity from the wrapped one [VH04]. This is similar to the sealer/unsealer pairs [Mor73] in the *E* language [Sti00, Mil06b].

Finalization

Finalization is a user-defined clean-up operation that is used for automatic memory management. When an entity is no longer reachable from an active part of the program, its memory can be reclaimed. Sometimes more than that has to be done for the program invariants to maintained. For instance, a data structure that counts the number of entities satisfying a particular property should be updated, or a file corresponding to a descriptor should be closed. Finalization handles cases such as these.

The current finalization in Oz does not guarantee that an entity that became unreachable is no longer used. The last operation performed on an entity before it becomes unreachable should truly be the last operation performed on the entity. To guarantee this, we propose to follow the “postmortem finalization” technique (executor of an estate), invented by Frank Jackson, Allan Schiffman, L. Peter Deutsch, and Dave Ungar (We found no reference to their work on this topic). When an entity becomes unreachable, the finalization algorithm invokes another entity, which plays the role of the executor of the first entity’s estate. The executor will perform all the clean-up actions but has no reference to the original entity.

10.3.2 Full Language

The full language has linguistic abstractions built on top of the kernel language and (base) libraries written in the full language itself. Linguistic support means that there is language syntax that is designed to support the abstraction. For example, a `for` loop can be given a concise syntax and implemented in terms of a `while` loop. We say that the `for` loop is a linguistic abstraction.

The full language has to be designed to support the writing of secure programs. This implies building new abstractions for secure programming and verifying that the current language satisfies the requirements for secure programming.

Modules and Functors

Like Oz, the full language Oz-E will provide operations to create and manipulate software components. In Oz, these components are values, called *functors*, and are defined through a linguistic abstraction. Functors are instantiated to become *modules*, which are executing entities. Modules are linked with other modules through a tool called the *module manager*. This linking operation gives authority to the instantiated module.

In Oz-E, the module manager has to be a tool for secure programming. For example, it should be easy to run an untrusted software component in an environment with limited authority, by linking it only to limited versions of running modules. Such modules can be constructed on the fly by the user's trusted shell or desktop program, to provide the right capabilities to host programs. This mechanism can also be used for coarse grained "sandboxing", e.g. to run a normal shell with a limited set of resources.

10.3.3 Environment Interaction

The security of Oz-E must be effective even though the operating system and network environments are largely outside of the control of the Oz-E application developers and language developers. How can this be achieved? In the long term, we can hope that the environment will become more and more secure and POLA compliant. In the short term, we need libraries that provide controlled access to the operating system and to other applications.

Security of an application ultimately derives from the user of the application. An application is secure if it follows the user's wishes. The user should have the ability to express these wishes via a user-friendly graphical user interface. Recent work shows that this can be done [Yee02]. For example, selecting a file from a browser window gives a capability to the application: it both designates the file and gives authority to perform an operation (such as *edit*) on the file. A prototype desktop environment, CapDesk, has been implemented using these ideas. CapDesk shows that both security and usability can be achieved on the desktop [SM02, Mil06b].

Oz has a high-level GUI tool called QTk. It combines the conciseness and manipulability of the declarative approach with the expressiveness of the procedural approach. QTk builds on the insecure module Tk and augments that functionality instead of restricting it. QTk has to be modified so that it satisfies the principles enunciated in [Yee02] and implemented in CapDesk.

10.4 Cross-Layer Concerns

The previous section presented a layered structure for the Oz-E language and system. In general however, security concerns cannot be limited to a single layer in such a structure. As explained in [MTS05], security concerns tend to be pervasive and not easily separable from functionality concerns in general.

In this section we discuss three concerns that affect all layers:

- Pragmatic issues of how to make the language easy for secure programming.
- Safe execution on distributed systems.
- The need for reflection and introspection.

10.4.1 Pragmatic Issues in Language Design

A secure language should not just make it *possible* to write secure programs, it must also make it *easy* and *natural*. Otherwise, one part of a program written with bad discipline will endanger the security of the whole program. The default way should always be the secure way. This is the security equivalent of fail-safe programming in fault-tolerant systems.

We propose to use this principle in the design of the Oz-E concurrency model. The two main concurrency models are message-passing concurrency (asynchronous messages sent to concurrent entities) and shared-state concurrency (concurrent entities sharing state through monitors). Experience shows that the default concurrency model should be message-passing concurrency. This is not a new idea; Carl Hewitt anticipated it long ago in the Actor model [Hew77, HBS73]. But now we have strong reasons for accepting it. For example, the Erlang language is used for building highly available systems [Arm03, AWWV96]. The *E* language is used for building secure distributed systems [MSC$^+$01, Mil06b]. For fundamental reasons, both Erlang and *E* use message-passing concurrency.

We therefore propose for Oz-E to have this default as well. One way to realize this is by the following semantic condition on the kernel language: *cells can only be used in one thread* (cells are the structures that provide mutable state). Applying this simple semantic condition would have as consequence that threads can communicate only through dataflow variables (declarative concurrency) and ports (message-passing concurrency).

10.4.2 Distributed Systems

The distribution model of Oz allows all language entities to be partitioned over a distributed system, while keeping the same semantics as if the entities were on different threads in a single system, at least when network or node failures are not taken into account. For every category of language entities (stateless, single-assignment, and stateful) a choice of distributed protocols is available that minimizes network communications and handles partial failure gracefully. Fault-tolerant abstractions can be built within the language, on top of this system.

We want to keep the Oz-E distribution system as close as possible to this model and put the same restrictions on communication with remote threads as with local threads (such restrictions were discussed in section 10.4.1).

The monolithic implementation of distribution in Mozart/Oz is currently being replaced by a modular implementation using the DSS (Distribution Subsystem) [BKB04, Kli05, KMV06]. The DSS is a language-independent library, developed primarily by Erik Klintskog and integrated into Oz by Boriss Mejias, that provides a set of protocols for implementing network-transparent and network-aware distribution.

This section briefly considers the opportunities offered by the DSS to add secure distribution to Oz-E.

Responsibility of the Language Runtime System

The division of labour between the DSS and the language system assigns the following responsibilities to the language runtime system:

1. Marshaling and unmarshaling of the language entities.
2. Differentiating between distributed and local entities.
3. Mapping of Oz-E entities and operations to their abstract DSS-specific types, which the DSS will distribute.
4. Choosing amongst the consistency protocols provided by the DSS, based on the abstract entity types, and adjustable for individual entities.

Secure marshaling should not break encapsulation, and every language entity should be allowed to specify and control its own distribution strategy and marshaling algorithm. *E* provides such marshaling support via “Miranda” methods that every object understands and that provide a safe default marshaling behavior which can be overridden.

Oz-E could build a similar implementation for the language entities that can perform method dispatching (e.g. objects). For the other entities (e.g. zero-argument procedures), Oz-E could allow specialized marshalers to be invited into the lexical scope of an entity when it is created. Section 10.5.2 gives two examples of how invitation can be implemented in Oz-E.

Alternatively, Oz-E’s kernel language could use only object-style procedures that by default forward marshaling behavior to marshalers, and that can override this behavior.

Depending on these choices, marshaling may need support at the kernel language level. The other three responsibilities of the language system can be provided as part of an Oz-E system library.

Responsibility of the Distribution Subsystem

The DSS itself takes responsibility for:

1. Distributing abstract entities and abstract operations.
2. Providing consistency, using the consistency protocols that were chosen.
3. Properly encrypting all communication, making sure that external parties cannot get inside the connection.
4. Ensuring that it is unfeasibly hard to get (guess) access to an entity without having received a proper reference in the legal way.

5. Authenticating the distributed entities to ensure that no entity is able to pretend to be some other entity.

In [BKB04] the DSS is shown to have security requirements that are compatible with the requirements for safely distributing capabilities. Three attack scenarios have been investigated:

1. Outsider attacks. It should be impossible (infeasibly hard) for an attacker node that does not have legal access to any distributed entities, to access an entity at a remote site or to make such an entity unavailable for legal access.

2. Indirect attacks. It should be impossible for an attacker node that has legal access to a distributed entity but not the one being attacked, to perform this kind of intrusion or damage.

3. Insider attacks. It should be impossible for an attacker node that has legal access to a distributed entity, to render the entity unavailable for legal access. This can only be guaranteed for protocols that do not distribute or relocate state such as protocols for asynchronous message sending or stationary objects (RPC), and only if the attacker node did not host the original entity.

Apart from the requirements of the second scenario, the current DSS implementation claims to follow all these requirements. DSS distribution protocols will be made robust to ensure that no DSS-node can be crashed – or forced to render entities unavailable for legal access – by using knowledge of the implementation. This is called "protocol robustification" and is still under development.

The fact that only asynchronous message sending and RPC-style protocols are protected from insider attacks is no objection for Oz-E. In section 10.4.1 such restriction was already put on the interaction between entities in different threads: normal threads on a single node will not be able to share cells.

10.4.3 Reflection and Introspection

To verify security properties at runtime, we propose to add the necessary primitive operations to the kernel language, so that it can be programmed in Oz-E itself. To what degree should an entity be able to inspect itself, to verify security properties? The problem is that there is a tension between introspection and security. For example, a program might want to verify inside a lexically scoped closure. Done naively, this breaks the encapsulation that the closure provides. In general, introspection can break the encapsulation provided by lexical scoping.

To avoid breaking encapsulation the *E* language allows a user-defined entity to invite relied-upon third parties (auditors) to inspect an abstract syntax tree representation of itself, and report on properties that they find. Section 10.5.2 shows with a code example how this could work in Oz-E.

Safe Debugging

In a distributed environment, where collaborating entities spread over different sites have different interests, how can debugging be done? The principle is similar to safe introspection: entities are in control of what debugging information they provide, and the debugger is a third party that may or may not be "invited into the internals" of the entity.

Code Verification

Loaded code should not be able to bring about behavior which exceeds behavior that could be described within the kernel language. Since we plan to use the Oz VM to run Oz-E bytecode, and the Oz VM itself provides no such guarantee, we must verify all code before loading it. Such verification of byte code is a cumbersome and error-prone task. Oz-E should therefore be restricted to load code from easily verifiable abstract syntax tree (AST) representations of kernel and full language statements, instead of byte code.

10.5 Some Practical Scenarios

In this section we take a closer look at how some of these ideas could be implemented. We want to stress that the examples only present one of the many possible design alternatives and do not express our preferences or recommendations. They are only provided as a clarification to the principles and as a sample of the problems that Oz-E designers will need to solve.

10.5.1 Implement Guards at what Level?

In section 10.2.2 we explained briefly the benefits of guards and how they are supported in E. Let us now show in pseudocode how expressions could be guarded in Oz-E and how a linguistic abstraction for guards could look like.

```
fun {EnumGuard L}
   if {Not {List.is L}}
   then raise notAList(enumGuard) end
   end
   for X in L do {Wait X} end
   proc {$ X}
      try
         if {Member X L}
         then skip
         else raise guardFailed(enumGuard) end
         end
      catch _ then
         raise guardFailed(enumGuard) end
      end
   end
end
Trilogic = {EnumGuard [true false undefined]}
{Trilogic (x == y)} % will succeed
{Trilogic 23}       % will raise an exception
```

Figure 10.3: A three valued logic type guard

The example in Figure 10.3 guards a three valued logic type consisting of **true**, **false**, or unknown. EnumGuard ensures that the set is provided as a list and that all its elements are bound. Then it creates a single parameter procedure that will do nothing if its argument is in the set, or raise an exception otherwise. A guard Trilogic is created from that, and tested in the two last lines. The first test will succeed, the second one will raise an exception.

What if we want to use this guard in a procedure declaration? Let us first assume we want to guard an input parameter, in this case X. Then:

```
proc {$ X:Trilogic ?Y} <S> end
```

can be translated into:

```
proc {$ X ?Y} {Trilogic X} <S> end
```

Guarding output parameters is more difficult. If Y is unbound then:

```
proc {P X ?Y:Trilogic} <S> end
```

can be translated as shown in Figure 10.4. Note that in Figure 10.4 the expression

```
proc {$ X ?Y}
   Y2
in
   thread
      try {Trilogic Y2} Y = Y2
      catch Ex
      then Y = {Value.failed Ex}
      end
   end
   <S>{Y->Y2}  %(1)
end
```

Figure 10.4: Guarding output parameters

marked (1) represents the statement <S> in which all free occurrences of the identifier Y are replaced by an identifier Y2 which does not occur in <S> (see chapter 13 of [VH04]).

These examples work for atomic values that are either input or output parameters, but they cannot simply be extended for guarding partial values, because the latter can be used for both input and output at the same time. Another problem is the relational programming style where all parameters can be input, output or both depending on how the procedure is used. This definitely calls for more research, which may possibly reveal the need for a new primitive to support guards.

10.5.2 A Mechanism for Invitation and Safe Introspection

Let us assume we have a new construct NewProc that takes an abstract syntax tree (AST) and an environment record mapping the free identifiers in the AST to variables and values, and returns a procedure. Instead of creating a procedure like this:

```
P1 = proc {$} skip end
```

we could now also create a procedure like this:

```
P1 = {NewProc ast(stmt:´skip´) env()}
```

To create an audited procedure, an auditor is invoked with an AST and an environment. The client of the procedure can call the auditor to inquire about the properties that it audits. Let's build an auditor to check declarative behavior. We first present one that keeps track of the declarative procedures it creates.

Figure 10.5 builds an auditor procedure that takes a message as argument. If the message matches createProc(...) it will investigate the AST and environment provided, and create a procedure by calling {NewProc ...} with the same arguments.

```
declare
local
   AuditedProcedures =  {NewCell nil}
   fun {Investigate AST Env}
      ... % return boolean indicating whether
          % {NewProc AST Env} returns a declarative procedure
   end
   proc {MarkOK P}   % remember that P is declarative
      AuditedProcedures := P | @AuditedProcedures
   end
   fun {IsOK P}      % checks if P is marked declarative
     {Member P @AuditedProcedures}
   end
in
   proc {DeclarativeAuditor Msg}
         case Msg
         of createProc(Ast Env ?P) then
            if {Investigate Ast Env}
            then
               NewP = {NewProc Ast Env}
            in
               {MarkOK NewP}
               P = NewP
            else P = {NewProc Ast Env}
            end
         [] approved(P ?B) then
            B = {IsOK P}
         end
      end
   end

P1 = proc {$} skip end
P2 = {DeclarativeAuditor createProc(ast(stmt:´skip´) env())}
P1OK = {DeclarativeAuditor approved(P1 $)} % P1OK will be false
P2OK = {DeclarativeAuditor approved(P2 $)} % P2OK will be true
```

Figure 10.5: Stateful auditor that investigates declarativity

If the investigation returned **true**, it will store the resulting procedure in a list of all the created procedures that succeeded the `Investigate` test. If the message matches `approved(...)` it will check this list.

Rees [Ree96] gives strong arguments against the approach of Figure 10.5, as it easily leads to problems with memory management, performance, and to semantic obscurity. For this reason W7 – like *E* – has chosen to provide a primitive function to create sealer-unsealer pairs. Figure 10.6 provides an alternative approach that avoids these drawbacks.

```
declare
local
   Secret = {NewName}
   fun {Investigate AST Env}
      ... % return boolean indicating whether
          % {NewProc AST Env} returns a declarative procedure
   end
   fun {MarkOK P}
      WrappedP in
      ... % wrap P in some sort of invokable chunk WrappedP
      ... % WrappedP when invoked, will transparently invoke P
      WrappedP.Secret = ok
      WrappedP
   end
   fun {IsOK P}  % checks if P is marked declarative
      try P.Secret == ok catch _ then false end
   end
in
   proc {DeclarativeAuditor Msg}
         case Msg
         of createProc(Ast Env ?P) then
            if {Investigate Ast Env}
            then P = {MarkOK {NewProc Ast Env $}}
            else P = {NewProc Ast Env}
            end
         [] approved(P ?B) then
            B = {IsOK P}
         end
      end
   end
```

Figure 10.6: Stateless auditor that investigates declarativity

The auditor built in figure 10.6 is stateless, and lets `MarkOK` wrap the created procedure in a recognizable entity that can be invoked as a normal procedure. An invokable *chunk* would do for that purpose, as it could have a secret field accessible by the *name* `Secret` known only to the auditor. For this to work, Oz-E's kernel language has to provide either invokable chunks or a primitive function to create sealer-unsealer functions.

Instead of providing the environment directly for the auditor to investigate, [Rei04] suggests a mechanism to manipulate the values in the environment before giving them to the auditor (e.g. by sealing) to make sure that they cannot be used for anything else than auditing.

Instead of inviting an auditor, one could invite a relied-upon third party that offers general introspection and reflection. It would have roughly the same code-frame as the auditor, but provide more detailed – and generally non-monotonic – information about the internal state and the code of the procedure.

10.6 Conclusions and Future Work

A long-term solution to the problems of computer security depends critically on the programming language. If the language is poorly designed, then assuring security becomes complicated. If the language is well-designed and consistently supports and promotes the principle of least authority, then assuring security becomes much simpler. With such a language, problems that appear to be very difficult such as protection against computer viruses and the trade-off between security and usability become solvable [Stib].

A major goal of Oz language research is to design a language that is as expressive as possible, by combining programming concepts in a well-factored way. The current version of Oz covers many concepts, but it is not designed to be secure. This chapter gave a rough outline of the work that has to be done to create Oz-E, a secure version of Oz that supports the principle of least authority and that makes it possible and practical to write secure programs.

Building Oz-E will be a major undertaking that will require the collaboration of many people. But the potential rewards are proportionally great. This initial investigation may be a starting point for future work by language designers who share this vision and want to participate in the project.

Chapter 11

Research Context and Agenda

The principles and ideas that were developed in this thesis have been applied only at a very small scale and in a limited context. In this chapter we situate this work in the broader context of related research, and we list the opportunities we think are the most important and appealing, for applying related and extended research in this broader context.

SCOLL concepts and technology were built upon foundations from a large volume of previous research work, performed by a growing community of software security researchers. Publications and experiences in the fields of capability based security, secure programming languages, and secure operating systems, were a crucial step-stone for our work. In comparison to the major achievements that were made in these areas of software security, our direct contributions can only be qualified as limited and of moderate importance.

Nevertheless, we are convinced that, when applied properly and developed to its full potential, our approach represents a crucial factor to:

1. Improve the practical applicability to software engineering, of the many existing results in software security.

2. Enable the formal investigation and comparison of alternative architectures and implementations of protection systems.

3. Enable the implementation of programming languages, environments and tools for provably secure programming.

4. Enable the design and implementation of software integration tools for provably secure integration at different levels of granularity: from complete applications, via components of different size and complexity, to the finest grained objects and procedures in the software.

5. Enable the implementation of provably secure operating systems.

6. Enable the practical and quantifiable application of the principle of least authority (POLA, Section 1.3.4) in software engineering and other fields.

11.1 Related and Useful Formalisms

This section discusses alternative formal approaches to safety analysis in software engineering, and interesting formal approaches to software analysis that may be used in the future to extend our work.

We restrict our scope to models that can consider individual software entities of arbitrary size and nature. Many formal safety models were built to reason about confinement between processes on the base of the user on whose behalf the processes run. These models are usually insufficiently expressive for our purpose, because we want to apply the principle of least authority at all levels in the software (Section 1.3.4).

Except for the KBM-based semantics of SCOLL, none the many formal models we encountered during our research provide a practical and meaningful way to aggregate subjects or to declare and refine subject behavior.

11.1.1 The (Extended) Schematic Protection Model

The relation between Schematic Protection Models (SPM) [San88] and Take Grant models is described in [Bis04]. SPM introduce a static notion of "protection type", which is assigned to a subject upon its creation, never to be changed thereafter. Like SCOLL and its KBM semantics (Chapters 5 and 6), SPM can therefore express rules for the propagation of authority that depend on the type of the subjects that are involved. A ticket in SPM is a description of a single right over another entity, similar to a capabilitiy in Take-Grant models.

From the tickets, link predicates can be constructed (using conjunction and disjunction but not negation) to express the permission-based preconditions of the rules. Based on the protection types, filter functions can be constructed to express extra preconditions for the propagation of tickets. Filter functions map couples of subjects to sets of tickets, to specify what tickets can be transferred from the first subject to the second one, when the link predicates are satisfied.

Like Take-Grant models, SPM provides direct support for the creation of new subjects of a certain type, only there can now be any number of types.

The expressive power of SPM, due to its powerful filter functions, allows SPM to express arbitrary SCOLL programs and KBM configurations. Because KBM configurations never grow in number of subjects, the converse is not true.

While SPM is a higher level model than HRU, and while it is also considerably more expressive than Take-Grant models, it was not conceived to make it easy to express subject behavior. Behavior based rules for the propagation of authority and permissions are very easily expressed in SCOLL, using the behavior predicates that can be declared in the language itself, and that can be refined when possible.

Bishop gives an example of collaboration under mutual distrust in [Bis04] to indicate an important drawback of SPM. To enable the collaboration between two untrusted parties, both parties create their own restricted proxy and introduce the proxies to each other, so as to interact via multiple indirections. To model this simple example that is very relevant to software engineers, a complex set of rights and manipulations has to be set up. Extended SMP (ESMP) is then introduced to solve this problem by providing support to model multi-parenthood: creation can now be a joint operation between multiple subjects.

While multi-parenthood can be an interesting idea at a certain level of abstraction, not many programming languages provide direct support for it. Moreover, most software engineers will not use it in connection with problems that can be solved easily with a couple of proxy patterns. The reader is invited to compare this approach to the natural and simple way in which section 8.3.1 (the membrane pattern) models a pattern that contains arbitrary many indirections and also proves it to be safe.

11.1.2 The Web-calculus

The web-calculus [Clo04] formally models the interaction of services and applications at an interface (specification) level. Its models are directed graphs in which the nodes represent static components, and the edges represent links (references) between them.

In the web-calculus the dynamic reference graph itself is the carrier of all state. While nodes themselves do not carry internal state, some of the edges do. The outgoing edges of a node are called “branches”. Branches can represent access channels through which the other nodes can be accessed, or closures with encapsulated state and a single anonymous method that can be invoked. The latter edges are labeled with a verb, the former with a noun.

Closures are invoked with a fixed input parameter (node or edge), and return a single output (node or edge. The invocation of a closure may grow the web (add new connected nodes to the graph) and/or reassign existing edges.

A small set of operations on edges is defined on the web-calculus, of which the most important ones are:

`GET`: retrieves the current target node of the edge. Its purpose is to provide graph inspection.

`POST`: invokes the closure associated with an edge on a list of input arguments. Its purpose is to provide graph mutation.

The web-calculus is an extension of the REST model (Representational State Transfer) [Fie00], to which the following elements are added:

1. a capability based security model
2. a model for the internal structure of a representation
3. a mechanism for defining the structure of a (web-) resources
4. a generic interface for distributed computation

The web-calculus principles are implemented in the Waterken$^{\text{TM}}$Server [Clo], an extensible HTTP server and web services platform.

It is interesting future work to investigate the relation between SCOLL and the web-calculus, to compare their respective possibilities, expressive power, and practical applicability in different situation, and to search for elements in either formal systems that can be assimilated to enrich the other one.

11.1.3 The Refinement Calculus

The refinement calculus [BvW98] is a formalization of the stepwise refinement method of program construction. The required behaviour of the program is specified as an abstract program which is then refined by a series of correctness-preserving transformations into an actual executable program.

Refinement is a transitive relation between the abstractions of a program. Refinement is also compositional, which allows subprograms and eventually statements to be refined separately. Specification statements consist of a precondition and a postcondition.

The refinement calculus has a set of laws that each describe a legal way to refine an abstract statement. It is always a legal refinement to strengthen the postcondition or to weaken the precondition. Other laws describe the introduction of certain constructs to the program, the elimination of certain constructs from it, the transformation of constructs and the composition of subprograms and statements.

The refinement calculator [BGL$^+$97] is a tool that supports the application of the refinement calculus to program development. The tool is built on the HOL theorem proving system [Bow] and has a practical graphical user interface with menus for transformations.

The refinement calculus focusses on preserving the correctness while refining from a more abstract model of the program to a less abstract model and eventually to a model that can easily be translated to a certain programming language. That translation step is not part of the refinement calculus.

The refinement calculus is applied to security verification in Java by the KindSoftware research group [KPC05].

We see several possible fields for future applications in SCOLL of the refinement calculus or similar calculi that may be derived from it:

1. The refinement of behavior predicates and the corresponding knowledge predicates were discussed in sections 5.5, 5.6.2 and 6.7.1. With minor adaptations, the refinement calculus could be applied to automate the process of behavior refinement.

2. The transformation of program code to SCOLL code could be automated and based on a set of laws similar to the refinement calculus laws. Only this time the transformation would go from more concrete to more abstract models, in steps that preserve safe approximation.

3. To provide concrete examples of safe patterns in SCOLL, the transformation of SCOLL code to program code could be automated, also based on a set of laws similar to the refinement calculus laws. This time the transformation would go from less concrete to more concrete models, using the reverse steps.

11.1.4 The Situation Calculus

The situation calculus is based on the work of Judea Pearl [Pea00] and formalizes the notions of causality and causal influence. The formalism allows us to express and handle counterfactual queries: questions asking whether event B could have occurred if event A had not occurred.

Causal Models

Causal models exist of two disjunct sets of variables, exogenous (U) and endogenous (V), and a set of functions F_X, for every variable $X \in V$, that map every interpretation of $V \setminus X$ to a value for X. The set of variables whose values affect the value of X is called the parent set of X, denoted PA_X.

Causal models are depicted in a causal diagram : a directed graph whose nodes correspond to the variables in $U \cup V$ and whose edges represent the parent relation.

If a particular interpretation for the exogenous variables is assumed, the causal model is called a causal world. It is assumed that in a causal word the values for the endogenous variables is determined by the interpretation for the exogenous variables. That is always the case if the causal diagram is acyclic.

Counterfactual queries are defined using the concept of submodel. A submodel of a causal model under the intervention $X = x$ defines a causal model where the function F_X is replaced by the constant function x.

Extensions in the Situation Calculus

The situation calculus [Hop02] extends the notion of causal model to include:

Objects : The variables represents objects in the real world, about whom the model will reason in a first order predicate logic.

Actions : Predicates, expressing relations between one or more objects to indicate events.

Situations : A sequence of actions, representing a possible world history.

Relational Fluents : Predicates whose last argument is a situation, to represent situation dependent predicates.

Functional fluents : Functions whose last argument is a situation, to represent situation dependent functions.

Initial Database Axioms : First-order situation calculus sentences which describe the initial state of the system.

Action Precondition Axioms : First-order situation calculus sentences which describe the conditions that must hold in order for an action to be permissible in a given situation.

Successor State Axioms : First-order situation calculus sentences which how fluents change in response to actions.

We find many intuitive similarities between SCOLL and the situation calculus, which is not very surprising since SCOLL was designed to reason about the upper bounds of authority, and authority can be described as causal influence between the entities in the model.

Objects loosely correspond to subjects, actions to behavior predicates, relational fluents to SCOLL's `goal` part, and initial database axioms to the `config` part. The action predication axioms can intuitively be interpreted as system rules, and the successor state axioms to behavior rules.

These vague and intuitive correspondences ask for a further investigation of the formal relations between SCOLL and the situation calculus. This may reveal interesting insights and show opportunities to expand SCOLL's expressive power, for instance to directly express safety concerns about what subjects should not cause what effects.

11.2 Approaches and Technologies to Optimize SCOLLAR

SCOLLAR is not merely used as a theorem prover, but also to find provably safe solutions to initial configurations and subject behavior. It was straight forward to implement this extension (second operation mode) because the tool was based on constraint programming technology. We only had to use the search facilities provided by the CP library.

The current version of SCOLLAR is a prototype implementation that can be used for relatively small patterns, but does not scale well. Several approaches and techniques can be applied to improve its performance and scalability:

Model optimization The performance of contraint programming tools is extremely sensitive to the way the problems are modeled. Our current experiences with the CP based implementations of SCOLLAR lead us to conclude that the approach using finite domain variables provides adequate performance for simple patterns.

The finite set approach is slower for small patterns, but its performance degrades less with increased size. The performance may benefit from a problem dependent choice between both approaches, or even from a mix of them.

Problem-specific optimization Given a problem and its representation as constraints in CP, many standard optimizations techniques can be applied. For instance, symmetry breaking is a technique to remove all solutions that are equivalent in some way. It can be applied in situations where unknown entities have access to each other, because their maximal behavior allows us to consider them as one unknown subject.

A generalization of this technique can be interpreted as partial-aggregation-on-the-fly : certain sets of predicates concerning certain sets of subjects can be aggregated and treated as one, as soon as the subjects have exceeded a certain limit of behavior and authority during the calculation.

Such optimizations can have an important impact on performance and scalability, because they reduce the depth of the search tree.

Graph constraints Chapter 9 discussed an alternative approach based on graph analysis, using the global DomReachability constraint over graphs [Que06b]. The approach is applicable to a specific subset of SCOLL patterns, in which the authority propagating behavior of the subjects can easily be expressed as a binary, transitive relation. The approach can provide important advantages when calculating solutions to the patterns in that subset.

Similar problem-specific approaches could be built into SCOLLAR to improve its overall performance and scalability.

Alternative CP libraries It is to be expected that the use of the GCode library [Sch] in Oz will improve performance with several orders of magnitude, as that library was designed for improved speed.

Alternatives to CP We feel that constraint programming is the most appropriate base technology for the problems SCOLLAR is solving, but there may be situations and patterns in which alternatives can improve performance and scalability.

We experimented briefly with classical SAT solvers [sat], but the approach did not yet provide promising results, mainly because these solvers require everything to be declared up front. Not only all predicate facts, but also all instantiations of the system rules and the behavior rules have to be declared before the computation can start.

Implementing a SAT solver that can instantiate rules and declare variables on demand would be an interesting idea for future research. It was shown that, in Oz, on-demand computation is part of the declarative subset [SCR03]. Therefore on-demand computation can be fitted seamlessly in declarative techniques like constraint programming.

11.3 Improved Expressive Power

11.3.1 More Expressive Behavior

In section 5.6.2 we already mentioned the possibility to provide support for refined predicates, given a general refinement lattice. Future extensions will improve the expressive power to express behavior by providing support in SCOLL for :

- data propagation,
- propagation of multiple arguments in a single invocation,
- non monotonic behavior changes,
- and behavior and knowledge inheritance.

11.3.2 More Expressive Goals

The dominator part of DomReachability (Chapter 9) can be of direct use to add expressive power to the safety goals in SCOLL. For instance, instead of simply stating that some effect (authority) should be prevented, we could instead require that all authority of a certain kind should only ever be available via a subject that we rely upon to revoke the authority in due time..

In the authority flow graph (to be derived from the access graph) a relied upon subject `alice` can revoke all `bob`'s authority over `carol`, if `alice` dominates `bob` in the authority-flow graph of which `carol` is the source.

This is an example of a more general approach we consider worth pursuing in the future: to loosen the strict correspondence between goals and predicates. While permission predicates, behavior predicates, and knowledge knowledge predicates are basic elements in SCOLL, there is no reason to limit the concerns for safety and liveness to those that can be expressed directly as conjunctions of predicates.

In fact, the limitation is unnatural and may force the user to model extra predicates and rules for the sole purpose of expressing and deriving the properties about the state that are crucial for the safety policy. Besides flow graph based constraints, all kinds of constraints can be considered, including:

- Sentences constructed from predicates and arbitrary logical connectives. For instance to express conditional safety concerns or disjunctions.
- Counterfactual constraints (see section 11.1.4) to directly express requirements about causality: who is not allowed to cause what effects in what way.

- Other constraints that correspond to safety properties, authority flow constraints, or counterfactual constraints in models that are derived from the situation can be used to express requirements concerning the history of the program state, revocability, repeatability, etc.

11.4 Opportunities for Integration

To improve the usability of SCOLL, it should be integrated into programming environments. Ideally the developer should be warned immediately if he writes or adapts code that renders his safety requirements unprovable. The developer would then have the choice to adapt his code or to ask the environment to suggest appropriate refinements of the predicates used in the model.

To integrate SCOLL as a tool in programming environments, we need to investigate approaches and techniques for the transformation from source code to SCOLL models and back. Figure 11.1 shows an overview of the components that are involved in this transformation.

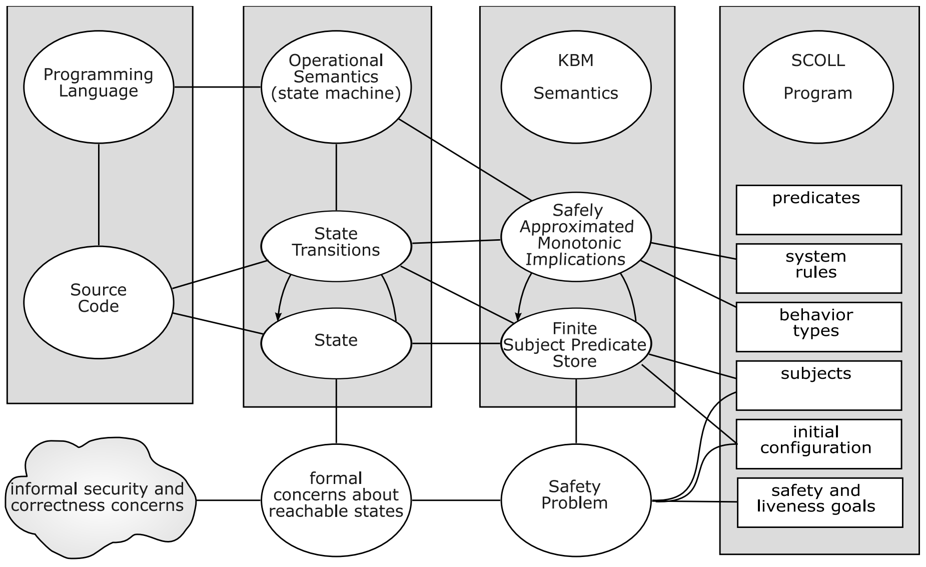

Programming Language
Operational Semantics (state machine)
KBM Semantics
SCOLL Program
Safely Approximated Monotonic Implications
predicates
State Transitions
system rules
Source Code
State
Finite Subject Predicate Store
behavior types
subjects
initial configuration
informal security and correctness concerns
formal concerns about reachable states
Safety Problem
safety and liveness goals

Figure 11.1: Components in the transformation between source code and SCOLL

Abstract Interpretation seems to be a promising technique for the transformation. Integrating the refinement calculus into such an approach could be very effective. Abstract interpretation of source code and SCOLL code would then result in a refinement calculus representation, that could be transformed in either directions as described in section 11.1.2.

11.4.1 SCOLL Carrying Code

In analogy to abstraction carrying code [HALGP05] and model carrying code [SRRS01], source code could also be adorned with a SCOLL model and a proof that the model safely approximates the actual code.

A simple and fast inspection of the proof would reveal its validity and whether the SCOLL model is indeed a safe approximation of the program. Integration of several such programs would result in a composed SCOLL model of which we can immediately inspect the safety properties.

The approach could provide automated acceptance or rejection of a software component to be integrated into existing software.

11.5 Other Opportunities and Applications

11.5.1 User Interface

Currently the graph visualization only works for the access graph. Only the binary permission *access* is represented in a graph. The user should have the possibility to visualize different aspects of the configuration and of the derived authority flow graphs.

11.5.2 Type Safety Analysis

In "Lightweight static capabilities" [KS06] Kiselyov and Shan propose a modular programming style that harnesses modern type systems to verify safety conditions. Their style has three ingredients:

1. A compact kernel of trust that is specific to the problem domain
2. Unique names (capabilities) that confer rights and certify properties, so as to extend the trust from the kernel to the rest of the application.
3. Static type proxies for dynamic values

They illustrate that their approach can provide statically provable type safety in dynamically typed languages. We have the strong impression that their proposed way of checking type safety corresponds to SCOLL patterns in which al entities of the same type are aggregated into a single subject they call the type proxy.

The relation between their approach and SCOLL needs further investigation, as it holds a possibility for another application area for SCOLL.

Chapter 12

Conclusions

When practicing secure programming, it is important to understand the restrictive influence programmed entities have on the propagation of authority in a program. To precisely model authority propagation in patterns of interacting entities, we introduced a new and practical formalism: Knowledge Behavior Models (KBM).

KBM allows reasoning about safety requirements in patterns of interacting entities, and has the required expressive power to be useful in secure software engineering. KBM provides an intuitive and expressive way to model, at an appropriate level of detail, the influence of the behavior of the entities on the maximal authority that can be deployed in a system.

We introduced the technique of aggregation to model all entities into a fixed and finite set of subjects and proved that aggregation results in safe approximations. Because of aggregation, KBM provides a tractable way to calculate a non trivial bound on behavior-based eventual authority.

The Safe Collaboration Language (SCOLL) is a new domain specific declarative language to express these patterns and to prove their safety or find maximal bounds for permissions and behavior that guarantee provably safe alternatives for the pattern. SCOLL supports an iterative, incremental approach to behavior model refinement. SCOLL can start with a safe but possibly too crude model and allows a gradual refinement of the model (while keeping it safe) when and where necessary. The language has a logical semantics that was expressed by means of KBMs.

To calculate the solutions for the safety problems expressed in SCOLL we have built SCOLLAR: a new tool for safety analysis, based on constraint logic programming and aimed at engineers and developers of secure software.

Several non-trivial examples showed the utility of the SCOLL based approach. Using SCOLL and SCOLLAR we showed that, by relying only on a programming language's runtime restrictions (e.g. capability safe languages) and on the behavior of strategically positioned entities, we can program many security policies that hold in the presence of arbitrary many untrusted entities.

The language and the tool encourage the investigation of additional or alternative safety mechanisms beyond capability-based protection systems, including mechanisms that perform access control by runtime reference monitoring. Its versatility makes the approach useful to illustrate and compare the safety and expressive power of alternative protection systems. This advantage is also interesting from a pedagogical point of view.

We expect that further development of this approach into a production-ready development aid will allow software providers and developers to take more, realistic, and well defined responsibility for the security of their code, including legal responsibility.

Publications

Publications arising from this thesis include:

Alfred Spiessens, Raphael Collet and Peter Van Roy (2003), Declarative Laziness in a Concurrent Constraint Language. *Proceedings of the 2nd International Workshop on Multiparadigm Constraint Programming Languages.* `http://uebb.cs.tu-berlin.de/MultiCPL03/`.

Fred Spiessens and Peter Van Roy (2005), The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language. *"Multiparadigm Programming in Mozart/Oz: Extended Proceedings of the Second International Conference MOZ 2004". Lecture Notes in Computer Science, Volume 3389.* Springer Verlag. Berlin. Germany.

Fred Spiessens and Peter Van Roy (2005), A Practical Formal Model for Safety Analysis in Capability-Based Systems. *Workshop on Trusted Global Computing 2005. Lecture Notes in Computer Science, Volume 3705, pages 248–278.* Springer Verlag. Berlin. Germany.

Yves Jaradin, Fred Spiessens, and Peter Van Roy (2005), SCOLL : A Language for Safe Capability Based Collaboration. *Research Report INFO-2005-10 Université catholique de Louvain.* Louvain-la-Neuve Belgium.

Fred Spiessens, Yves Jaradin, and Peter Van Roy (2005), Using Constraints To Analyze And Generate Safe Capability Patterns. *First International Workshop on Applications of Constraint Satisfaction and Programming to Security (CPSec'05). Research Report INFO-2005-11 Université catholique de Louvain.* Louvain-la-Neuve Belgium.

Fred Spiessens, Yves Jaradin, and Peter Van Roy (2005), SCOLL and SCOLLAR: Safe Collaboration based on Partial Trust. *Research Report INFO-2005-12 Université catholique de Louvain.* Louvain-la-Neuve Belgium.

Bibliography

[AP67] William B. Ackerman and William W. Plummer. An implementation of a multiprocessing computer system. In *SOSP '67: Proceedings of the first ACM symposium on Operating System Principles*, pages 5.1–5.10, New York, NY, USA, 1967. ACM Press. 4

[Arm03] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology (KTH), Stockholm, December 2003. 10.4.1

[AWWV96] Joe Armstrong, Mike Williams, Claes Wikström, and Robert Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1996. 10.4.1

[BGL$^+$97] Michael Butler, Jim Grundy, Thomas Långbacka, Rimvydas Rukšėnas, and Joakim von Wright. The refinement calculator: Proof support for program refinement. In Lindsay Groves and Steve Reeves, editors, *Formal Methods Pacific'97: Proceedings of FMP'97*, Discrete Mathematics and Theoretical Computer Science, pages 40–61, Wellington, New Zealand, July 1997. Springer-Verlag. 11.1.3

[Bis81] Matt Bishop. Hierarchical take-grant protection systems. In *Proceedings of the eighth ACM symposium on Operating systems principles*, pages 109–122. ACM Press, 1981. 3

[Bis04] Matt Bishop. *Computer Security : Art and Science*. Addison Wesley, September 2004. 1.3.1, 2.1.1, 11.1.1

[BKB04] Zacharias El Banna, Erik Klintskog, and Per Brand. Report on security services in distribution subsystem. Technical Report PEPITO Project Deliverable D4.4 (EU contract IST-2001-33234), K.T.H., Stockholm, January 2004. 10.4.2, 10.4.2

[BL74] D.E. Bell and L. LaPadula. Secure Computer Systems. In *ESD-TR*, pages 83–278. Mitre Corporation, 1974. Electronically available at: `http://www.albany.edu/acc/courses/ia/classics/belllapadula1.pdf`. 9.4.3

[Boe84] W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proceedings of 7th DoD/NBS Computer Security Conference*, pages 45–54, September 1984. http://zesty.ca/capmyths/boebert.html. 3.4.4, 4.1, 4.5, 8.4, 8.4.2

[Bow] Jonathan Bowen. The hol theorem prover. `http://vl.fmnet.info/hol/`. 11.1.3

[BS79] Matt Bishop and Lawrence Snyder. The transfer of information and authority in a protection system. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 45–54. ACM Press, 1979. 2.1.1, 3, 3.2, 3.2.1, 3.3.2

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science. 11.1.3

[Cap] Cap-talk mailing list. `http://www.eros-os.org/mailman/listinfo/cap-talk`. 4.2

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991. 10.2.2

[Clo] Tyler Close. Waterken $^{\text{TM}}$server. `http://www.waterken.com/dev/Server/`. 11.1.2

[Clo04] Tyler Close. Web calculus. `http://www.waterken.com/dev/Web/`, 2004. 11.1.2

[CMP00] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly & Associates, Paris, 2000. 5.1

[DH65] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. Technical Report MIT/LCS/TR-23, M.I.T. Laboratory for Computer Science, 1965. 4, 4.1, 2, 8.4.3, 9.4.4, 10.2.2

[Dij82] Edsger W. Dijkstra. *On the Role of Scientific Thought*. Springer Verlag, 1982. 8.5.4

[FA03] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Cognitive Technologies. Springer, Sept 2003. 7.6

[FB96] Jeremy Frank and Matt Bishop. Extending The Take-Grant Protection System, December 1996. Available at: `http://citeseer.ist.psu.edu/frank96extending.html`. 3, 3.2.4

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, California, 2000. 11.1.2

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994. 5.5.5, 10.2.2

[GJ79] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the The Theory of NP-Completeness*. W. H. Freeman and Company, 1979. 9.4.1

[GM78] Herve Gallaire and Jack Minker, editors. *Logic and Data Bases*. Perseus Publishing, 1978. 2.1.5, 5.7.2

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000. 8.1.4

[Gon89] Li Gong. A Secure Identity-Based Capability System. In *IEEE Symposium on Security and Privacy*, pages 56–65, 1989. 4.1, 4.5, 8.4

[GV05] GraphViz - Graph Visualization Software, 2005. `http://www.graphviz.org/`. 2.3.2, 8.1.4

[HALGP05] Manuel V. Hermenegildo, Elvira Albert, Pedro L&#243;pez-Garc&#237;a, and Germ&#225;n Puebla. Abstraction carrying code and resource-awareness. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 1–11, New York, NY, USA, 2005. ACM Press. 11.4.1

[Har85] Norman Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985. 4

[Har88] Norman (Norm) Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988. 4.6, 8.1.1, 8.1.1

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245, August 1973. 10.4.1

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977. 10.4.1

[HKN05] Shai Halevi, Paul A. Karger, and Dalit Naor. Enforcing confinement in distributed storage and a cryptographic model for access control. Cryptology ePrint Archive, Report 2005/169, 2005. 4.1, 4.5, 8.4

[HLS05] P.J. Hawkins, V. Lagoon, and P.J. Stuckey. Solving set constraint satisfaction problems using robdds. *Journal of Artificial Intelligence Research*, 24:109–156, 2005. 9.5.1

[Hop02] M. Hopkins. Causality and counterfactuals in the situation calculus. Tech Report R-301, UCLA Cognitive Systems Laboratory, 2002. 11.1.4

[HR78] Michael A. Harrison and Walter L. Ruzzo. Monotonic protection systems. *Foundations of Secure Computation*, pages 337–365, 1978. 1.3.1

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976. 2.1, 3, 3.1.2, 3.1.3, 3.1.3, 3.1.4, 3.1.5, 4

[HS06] Peter Hawkins and Peter Stuckey. A hybrid bdd and sat finite domain constraint solver. In *PADL 2006 Proceedings*, volume 3819 of *Lecture Notes in Computer Science*. Springer, 2006. 9.5.1

[JLS76] A. Jones, R. Lipton, and L. Snyder. A linear-time algorithm for deciding security. In *Proceedings of the 17th Symposium on Foundations of Computer Science*, pages 33–41, sep 1976. 3

[KGRB02] Alan H. Karp, Rajiv Gupta, Guillermo Rozas, and Arindam Banerji. Split Capabilities for Access Control. Research Report HPL-2001-164R1, HP Labs, Palo Alto, California, June 2002. Available at http://www.hpl.hp.com/techreports/2001/HPL-2001-164R1.html. 3.4.1

[KL87] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. *IEEE Trans. Softw. Eng.*, 13(2):202–207, 1987. 4.1, 4.5, 8.4

[Kli05] Erik Klintskog. *Generic Distribution Support for Programming Systems*. PhD thesis, KTH Information and Communication Technology, Sweden, 2005. 10.4.2

[KMV06] Erik Klintskog, Boris Mejías, and Peter Van Roy. Efficient distributed objects by freedom of choice. In *Revival of Dynamic Languages, ECOOP'06*, July 2006. 10.4.2

[KN93] Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot*. Murray Hill, NJ, 1993. 8.1.4

[KPC05] Joe Kiniry, Erik Poll, and David Cok. Fm 2005 tutorial : ”design by contract and automatic verification for java with jml and esc/java2”. `http://secure.ucd.ie/documents/tutorials/fm05.html`, 2005. 11.1.3

[KS06] Oleg Kiselyov and Chun-Chieh Shan. Lightweight static capabilities. August 2006. Workshop “Programming Languages meets Program Verification” at the 2006 Federated Logic Conference. 11.5.2

[Lam73] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973. 2.1.4, 3.2.5, 8.3

[Lan06] Charles Landau. Mail subject: Ambient authority in DVH, July 2006. `http://www.eros-os.org/pipermail/cap-talk/2006-July/005504.html`. 4.2

[lit] Little Snitch ®. Commercial Software available at `http://www.obdev.at/products/littlesnitch/`. (document), 8.1.5

[LS77] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977. 3.2.4

[LT79] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979. 9.2.1

[McK] Matthew McKeon. The Internet Encyclopedia of Philosophy. `http://www.iep.utm.edu/l/logcon.htm`. 5.7.1

[Mil] Mark. S. Miller. The Confused Deputy. `http://www.erights.org/elib/capability/deputy.html`. 8.1.1

[Mil03] Mark S. Miller. Building a Virus-Safe Computing Platform: Don't Add Security, Remove Insecurity, nov 2003. Talk given at the Information Theory Seminar of Hewlett Packard Laboratories, and at the Stanford Computer Systems Laboratory Colloquium. `http://www.cypherpunks.to/erights/talks/virus-safe/dont-add.ppt`. 10.1.3

[Mil06a] Mark S. Miller. Mail subject: Boebert attacks, capability review, July 2006. `http://www.eros-os.org/pipermail/cap-talk/2006-July/005473.html`. 8.4.3

[Mil06b] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. 1.3.6, 4.1, 4.1.6, 4.3, 5.1, 8.2.1, 8.2.1, 8.4, 9.4.4, 10.2.2, 10.2.2, 10.2.2, 9.4.3, 10.3.3, 10.4.1

[Mor73] James H. Morris. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973. 9.4.3

[Moz03] Mozart Consortium. The Mozart Programming System, version 1.3.0, 2003. Available at `http://www.mozart-oz.org/`. 5.1, 7, 7.6.5

[MS03] Mark S. Miller and Jonathan Shapiro. Paradigm Regained: Abstraction Mechanisms for Access Control. In *8th Asian Computing Science Conference (ASIAN03)*, pages 224–242, December 2003. 1.3.3, 1.3.8, 3.4.2, 3.4.4, 4.1, 4.3, 10.2.3

[MSC$^+$01] Mark S. Miller, Marc Stiegler, Tyler Close, Bill Frantz, Ka-Ping Yee, Chip Morningstar, Jonathan Shapiro, Norm Hardy, E. Dean Tribble, Doug Barnes, Dan Bornstien, Bryce Wilcox-O'Hearn, Terry Stanley, Kevin Reid, and Darius Bacon. E: Open source distributed capabilities, 2001. Available at `http://www.erights.org`. 1.3.4, 4, 5.1, 8.2.1, 10.1.1, 10.1.3, 10.4.1

[MTS05] Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro. The Structure of Authority: Why Security is Not a Separable Concern. In *Multiparadigm Programming in Mozart/Oz: Proceedings of MOZ 2004*, volume 3389 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005. 4.6, 8.5.4, 10.4

[MYS03] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Draft available at `http://zesty.ca/capmyths`, 2003. 1.3.2

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972. 2

[Pea00] Judea Pearl. *Causality. Models, Reasoning and Inference*. Cambridge University Press, 2000. 11.1.4

[Que06a] Luis Quesada. The bounded transitive closure problem, 2006. Available at *http://www.info.ucl.ac.be/˜luque/papers/btc.pdf*. 9.4.1

[Que06b] Luis Quesada. *Solving Constrained Graph Problems using Reachability Constraints based on Transitive Closure and Dominators*. Doctoral dissertation, Université catholique de Louvain, Louvain-la-Neuve, Belgium, September 2006. 9, 11.2

[QVD05] Luis Quesada, Peter Van Roy, and Yves Deville. The reachability propagator. Research Report INFO-2005-07, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2005. 9

[QVDC06] Luis Quesada, Peter Van Roy, Yves Deville, and Raphaël Collet. Using dominators for solving constrained path problems. In *PADL 2006 Proceedings*, volume 3819 of *Lecture Notes in Computer Science*. Springer, 2006. 1.2, 9

[Red74] David D. Redell. *Naming and Protection in Extendable Operating Systems*. PhD thesis, Cambridge, MA, USA, 1974. 2.2.1, 8.2.1, 10.2.3

[Ree96] Jonathan A. Rees. A security kernel based on the lambda-calculus. Technical report, MIT, 1996. 4, 4.1.1, 10.5.2

[Rei04] Kevin Reid. [e-lang] Proposal: Auditors without unshadowable names, August 2004. Mail posted at e-lang mailing list, available at `http://www.eros-os.org/pipermail/e-lang/2004-August/010029.html`. 10.5.2

[San88] Ravinderpal Singh Sandhu. The schematic protection model: its definition and analysis for acyclic attenuating schemes. *J. ACM*, 35(2):404–432, 1988. 3, 3.2.3, 3.3.3, 11.1.1

[sat] Sat live ! `http://www.satlive.org/`. 11.2

[Sch] Christian Schulte. generic constraint development environment. `http://www.satlive.org/`. 11.2

[Sch02] Christian Schulte. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002. 2.3, 5.7.2, 7, 7.6, 7.6.5

[SCR03] Alfred Spiessens, Raphaël Collet, and Peter Van Roy. Declarative laziness in a concurrent constraint language. In *2nd International Workshop on Multiparadigm Constraint Programming Languages*, pages 5–18, September 2003. 7.6.2, 11.2

[SDN+04] Jonathan Shapiro, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark S. Miller. Towards a Verified, General-Purpose Operating System Kernel. Technical report, Johns Hopkins University, 2004. Available at `http://www.coyotos.org/docs/osverify-2004/osverify-2004.pdf`. 4

[SGL97] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems*, 19(2):239–252, March 1997. 9.2.1

[SJV05] Fred Spiessens, Yves Jaradin, and Peter Van Roy. Using Constraints To Analyze And Generate Safe Capability Patterns. Research Report INFO-2005-11, Département d'Ingénierie Informatique, Université catholique de Louvain, Louvain-la-Neuve Belgium, 2005. Presented at CPSec'05. Available at `http://www.info.ucl.ac.be/~fsp/rr2005-11.pdf`. 7.7.2

[SM02] Marc Stiegler and Mark S. Miller. A Capability Based Client: The DarpaBrowser. Technical Report Focused Research Topic 5 / BAA-00-06-SNK, Combex, Inc., June 2002. Avalalbe at `http://www.combex.com/papers/darpa-report/`. 1, 10.2.3, 10.3.3

[SM06] Marc Stiegler and Mark S. Miller. How Emily Tamed the Caml. Research Report HPL-2006-116, HP Labs, Palo Alto, California, Aug 2006. Available at http://www.hpl.hp.com/techreports/2006/HPL-2006-116.html. 5.1

[Smo95] Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995. 7.6.5

[Spi] Fred Spiessens. confused deputy. `http://everything2.com/index.pl?node=confused%20deputy`. 4.6, 8.1.1

[SRP91] Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Principles of Programming Languages (POPL)*, pages 333–352, Orlando, FL, January 1991. 7.6

[SRRS01] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Model-carrying code (mcc): a new paradigm for mobile-code security. In *NSPW '01: Proceedings of the 2001 workshop on New security paradigms*, pages 23–30, New York, NY, USA, 2001. ACM Press. 11.4.1

[SS73] Jerome H. Salzer and Michael D. Schroeder. The protection of information in computer systems. In *Fourth ACM Symposium on Operating System Principles*, March 1973. 1.3.4, 8.3

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999. 4

[Stia] Marc Stiegler. The Confused Deputy. `http://www.skyhunter.com/marcs/capabilityIntro/confudep.html`. 8.1.1

[Stib] Marc Stiegler. The SkyNet Virus: Why it is Unstoppable; How to Stop it. Talk available at `http://www.erights.org/talks/skynet/`. 10.6

[Sti00] Marc Stiegler. *The E Language in a Walnut*. 2000. Draft available at `http://www.erights.org`. 10.2.2, 9.4.3

[SV05] Fred Spiessens and Peter Van Roy. The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language. In *Multiparadigm Programming in Mozart/Oz: Extended Proceedings of the Second International Conference MOZ 2004*, volume 3389 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005. 2.2.2, 5.1, 10.1.3

[Tar83] Alfred Tarski. *On the concept of logical consequence*. Hackett Publishing, Indianapolis, 1983. 5.7.1

[Tur37] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 43 of *2*, pages 544–546, London, UK, 1937. the London Mathematical Society. 3.1.4, 4

[VD91] Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 745–759. MIT Press, 1991. 9.5.1

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004. 5.1, 6, 7.6.5, 10.2.1, 10.2.2, 10.3, 10.3.1, 9.4.3, 10.5.1

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for java. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 116–128. ACM Press, 1997. 3, 4.1, 4.5, 4.6, 8.1.2, 8.4

[wik] Confused deputy problem. `http://en.wikipedia.org/wiki/Confused_deputy_problem`. 8.1.1

[WPJV03] Bart De Win, Frank Piessens, Wouter Joosen, and Tine Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. In *ACSA Workshop on the Application of Engineering Principles to System Security Design - Final Report (Serban, C., ed.)*, pages 1–10, September 2003. 8.5.4

[Yee02] Ka-Ping Yee. User interaction design for secure systems. In *4th International Conference on Information and Communications Security (ICICS 2002)*, 2002. UC Berkeley Technical Report CSD-02-1184. 10.3.3

[YM00] Ka-Ping Yee and Mark S. Miller. Auditors: An extensible, dynamic code verification mechanism. Available at `http://www.erights.org/elang/kernel/auditors/`, 2000. 10.2.2