A Practical Formal Model for Safety Analysis in Capability-Based Systems

Fred Spiessens and Peter Van Roy

Université catholique de Louvain
Louvain-la-Neuve, Belgium
{fsp,pvr}@info.ucl.ac.be

Abstract. We present a formal system that models programmable abstractions for access control. Composite abstractions and patterns of arbitrary complexity are modeled as a configuration of communicating subjects. The subjects in the model can express behavior that corresponds to how information and authority are propagated in capability systems. The formalism is designed to be useful for analyzing how information and authority are confined in arbitrary configurations, but it will also be useful in the reverse sense, to calculate the necessary restrictions in a subject's behavior when a global confinement policy is given.
We introduce a subclass of these systems we call "saturated", that can provide safe and tractable approximations for the safety properties in arbitrary configurations of collaborating entities.

1 Introduction

Since Harrisson, Ruzzo, and Ullman (HRU) showed in 1976 [HRU76] that safety properties are generally intractable, two approaches have been explored to calculate a safe approximation for safety properties. The first one is to keep on using Turing Complete models and to deal with the intractability by limiting the resources allocated to the safety checker. The checker will "give up" after exhausting the given resources, and report the possibility of a safety breach without proof. Such an approach can for instance be implemented in the SPIN model checker [Hol97]. This allows the user of the model checker to iteratively increase the precision (depth) of the calculation.

A second approach builds tractability into the model: instead of calculating a finite approximation of a possibly intractable safety property, it tries to calculate the exact value of the corresponding tractable safety property in an approximate model. Take-Grant systems [BS79] are an example of this approach, in which the safety properties are tractable [LS77,FB96]. This is the approach we take in this paper. Because checking tractable models can take arbitrary many resources too, we will take care that the approximation can be easily adapted: coarsening the model in some regions to make it simpler while refining it in other regions to gain precision.

Regardless of the approach taken, model checking involves the translation from a real world situation to a configuration in the formalism, and from the

calculated safety properties to conclusions that can be applied to the actual problem. Both translations should be well understood by the user of the model checker and should be explicit and well documented.

To ensure that the formalism is practical and useful to software engineers, we aimed for these translations to be easily described in terms of programming and design properties. We want our formalism to be useful at all levels of abstraction, during all stages of the software building process. The precision of modeling can be iteratively adapted. The resulting formal system forms a suitable base for the implementation of a dedicated model checker.

We developed Authority Reduction Systems via a series of consecutive refinements starting from Take-Grant systems [BS79]. The structure of this paper coarsely reflects this history. We first give an introduction to capability based security in Section 2. As a running example, we describe in Section 3 a simple pattern of authority delegation and revocation, called the Caretaker [MS03]. We will use this pattern as a touch stone for the expressive power of our formalism.

From studying capabilities [DH65] in general, and especially from the clarifications about capability based security recently provided by Miller and Shapiro [MS03], we concluded that modeling *collaborative behavior* is crucial when modeling capabilities accurately. When propagating authority from one subject to another, the authority reducing behavior of the subjects involved should be taken into account.

We explain in Section 4 that this collaborative aspect is underdeveloped in classical Take-Grant configurations, where only two kinds of subject behavior are considered: active vs. passive. We then describe three consecutive steps to refine this formalism. We present every step in its own section: Sections 5 to 7. Every consecutive refinement will build upon the previous one: avoiding its drawbacks and adding expressive power where necessary while keeping the safety properties tractable.

As a first step, we model collaborative behavior in Section 5 by annotating every subject with a set of properties. Each property describes three orthogonal aspects of collaborative behavior:

- the possibility of *initiating* a collaboration (invoking behavior) vs. *responding* (being invoked)
- the possibility of exchanging *capabilities* vs. *data* (information)
- the possibility of providing something during the collaboration (the emitting subject or *emitter*) vs. accepting something (the collecting subject or *collector*).

These three orthogonal aspects result in eight distinctive properties (e.g. possibly *initiate* the *emitting* of *data*), the combination of which allowed us to model 256 different types of behavior, including both types that are available in Take-Grant systems. While the resulting formalism had gained considerable expressiveness, it soon became clear that to model many relevant problems and patterns further refinement would be required.

We tried several approaches to make behavior compositional (to build arbitrarily complex subject behavior from configurations of simple subjects), and

we present the most important one in Section 6. This is step two in our effort to gain expressive power. The approach is well fit to model composite entities like components and modules. However, this step did not completely meet our "practicality" requirement at the finest grained level. The collaborative behavior of smaller entities like objects or procedures is often *not* structured as a configuration of collaborating sub-entities with simpler collaborative behavior. Our model had to allow us to express more refined behavior *directly*.

In Section 7 (step three) we show how to express a subject's behavior in terms of its relations to other subjects it has access to. We think this approach will provide a practical way to model security related problems in software engineering. Variables in the scope of procedures or objects often *do* correspond to relations with other program entities.

The behavior can now express how *future relations* will be decided too, concerning subjects that will be acquired through collaboration. This is relevant in practice since it can be deduced from static analysis how entities acquired via invocation will be treated: some will be stored into a variable with limited scope, some will be used as arguments in consecutive invocations, and some will be invoked. We are not trying to accurately model relations between subjects for its own sake however. We just want to model a more precise approximation of an entity's collaborative behavior by taking (part of) its relations into account.

We discuss similar use of predicates and logic programming in other research work on security (Section 8) and give an overview of what remains to be done in Section 9.

A streaming video presentation on the contents of this paper is available [SV05b].

2 Capability Security and Capability Secure Languages

The security concept we call a *capability* was introduced by Dennis and Van Horn [DH65] in the 1965. The concept is very simple: make designation unforgeable and combine it with authority, then you have a capability. If you are able to reference an entity (or a resource), you are allowed to use it. On first sight this can seem a very weak and discretional policy, but a quick exploration of the consequences will correct this impression.

We have to define the concept of *authority* first. Authority is the influence a program entity can have on other program entities and on the "system" in general. Part of this influence can be through the redistribution of information (data), and part of it can be about redistributing authority itself amongst the program entities. Potential authority is the whole of effects a subject could possibly induce if it would use its capabilities to the largest possible extent. Actual authority is the part of Potential Authority that is actually used by the entity. Actual authority takes the known restrictions in an entity's collaborative behavior into account. We use the term *Authority Reduction* to indicate the difference between a entity's potential authority and its actual authority.

In capability systems, all authority is carried via capabilities and capabilities can be distributed in four ways:

By Initial Conditions: We start reasoning from a given configuration in which some entities have access to some other entities. Since access is via references and all references are unforgeable capabilities, all entities are only referred to by (via, as) capabilities.

By Parenthood: An entity can create another entity, and by the act of creation get access to the created entity.

By Endowment: The created entity is endowed with (part of) the parent's authority. The parent decides which part.

By Invocation: Alice can introduce Bob to Carol by invoking Carol with Bob as an argument, but only if Alice has access to Bob and to Carol. This mechanism is sometimes referred to as "granting". If on the other hand Carol has access to Alice and Alice to Bob, then Carol can invoke Alice, and Alice can return Bob as the result of the invocation. This is sometimes referred to as "taking".

Ambient authority is all authority that can be acquired in any other way. Capability systems completely avoid ambient authority. Invocation is the most important (and potentially dangerous) way of authority propagation. Keep in mind though, that both the invoker and the invoked entity have control. The principle is also called object-capabilities, because the encapsulation of authority resembles the encapsulation of data, and the control that can be exerted by the invoked entity resembles the invocation of a method. Instead of setting up an access control policy separated from the functionality of a program, the programmer controls capability propagation by carefully controlling what entities will invoke what other entities and what will be the input and output arguments. This is not always a simple task, but a well designed capability secure programming language can help [SV05a].

Let us see how capabilities score on the security checklist compiled by Salzer and Schroeder [HS73] :

Least Authority: This is the principle of least authority (POLA) at which capabilities excel. No ambient authority is provided and no authority is ever granted implicitly. Instead of granting coarse grained privileges or rights, capabilities are created to *fit almost exactly* the least authority an entity needs. Even if the right to use and pass a capability is eternal and absolute, one-shot authority, temporal authority, revocable or conditional authority can all be programmed into a capability.

Simplicity of Mechanism: No other mechanisms than referencing and invocation are necessary to propagate and control authority. The mechanism to enforce the capability rules (at the base of every programmed policy) is simple and universal, and limits the way how capabilities can be distributed, and how data can be overtly distributed. That makes the reference monitor so simple that it will usually be a part of the language runtime.

Complete Mediation Because of the necessity for collaboration when exerting or propagating authority, the actual authority provided by the invocation can be very dynamic and can change with every invocation. Without the need for managing the validity or expiration of a capability, the invoked entity can completely control the actual authority it provides, based on the circumstances it can observe and on what it can learn from the arguments it is invoked with.

Least Common Mechanism: Another property at which capabilities excel: the authority provided by every capability is programmed and can react to what it can observe of its environment (local state, parameters, etc.). Of course, invocations of the same procedure or method share the static part of their behavior (by having the same code) but to have capabilities share their authority policies beyond this obvious lower bound would actually be hard to accomplish, and because of the scalability of the design, there is certainly no need to do so.

Tamper Resistance: This burden is on the language designers. They have to make sure that no holes exist through which ambient authority becomes available. If done well, capabilities can even prevent confused deputy attacks. A deputy is an entity to which its clients have to delegate authority to enable it to perform a service on their behalf. A confused deputy is a deputy that does not know the difference between its own authority and the authority delegated by the client, so that it can be lured into using its own authority on behalf of a faking client. As explained in [Har89,SV05a], capabilities can easily avoid that vulnerability.

Scalability: Since references are necessary anyway, combining them with authority does not in the least affect scalability, even as the authority is managed at the finest grained level.

But there are drawbacks too:

Ease of Use: Instead of carelessly giving out access to unknown entities, the programmer has to consider very carefully what the least authority is he should provide. Giving less will introduce bugs, giving more will introduce unnecessary vulnerabilities. This is certainly *not an easy task*. Therefore a real capability secure programming language should have no mechanisms that make this task even harder [SV05a]. But since capabilities are the only mechanism that can actually prevent confused deputy attacks [Har89], the task becomes feasible at last. It would seem that actually enforcing a security policy (not just being able to declare it) is never an easy task indeed.

Open Design: Attackers should be allowed to inspect the code base in which they will inject their malicious entity. They then have the same weapons as the programmer to search for security holes. That means that the programmer's weapons should be nearly perfect, but they definitely are not, as code analysis is a hard task, even in well designed capability languages. This topic reveals the *need for tool-supported formal safety analysis*. This is the main rationale for our contribution in this paper: to provide a simple but powerful

and practical formal model for authority propagation that can be the basis for such a tool.

Orthogonality of Concerns: The security policy is completely entangled with the functionality, programmed together into the same methods and procedures. In "The Structure of Authority: Why Security is Not a Separable Concern" [MTS05] Miller, Tulloh, and Shapiro explain the deeper reasons for this intrinsic entanglement of concerns. It remains to be investigated whether this unavoidable burden can somehow be relieved.

3 A Running Example: The Caretaker Pattern

Throughout this paper we will refer to a pattern of capability propagation and revocation, called the Caretaker. The pattern is useful and used in practice when programming revocable authority in capability secure language [MSC+01]. Consider a configuration of five subjects, having access to each other as indicated by the arrows (first part of Figure 1). As usual in a capability system, access and right-of-invocation are combined.

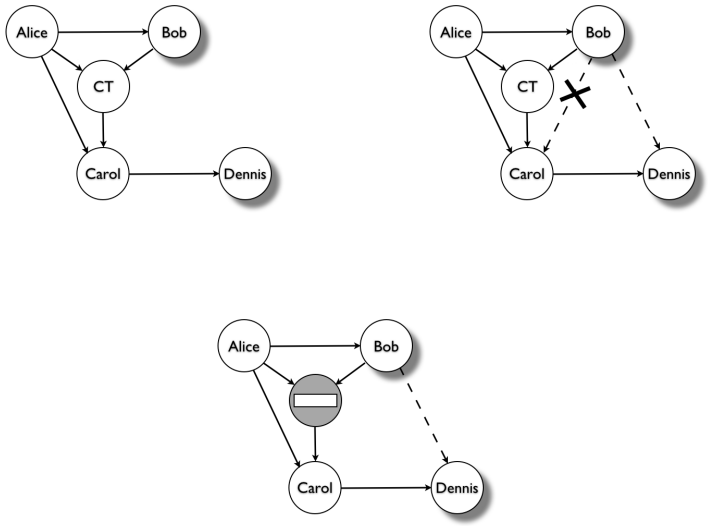


Fig. 1. The Caretaker in different stages

Alice wanted to give Bob revocable authority to use Carol, and therefore created a Caretaker entity (CT) that will proxy for Carol, and provided CT to Bob. Dennis depicts some authority Carol does not mind sharing with Bob.

Bob and Dennis are subjects of which we do not model any knowledge about their behavior, and therefore have to assume that they might use all authority they have. We indicate "unknown" subjects with a shadow, and access that *could possibly* be gained via collaboration with a dashed arrow. The second part of Figure 1 shows what access is expected to be propagated by collaboration (dashed arrow), and what access should be impossible (dashed arrow with cross).

Alice can instruct CT to stop being a proxy for Carol, thereby becoming opaque to authority propagation (the stop sign in the last part of the figure), and depriving Bob of the authority to further use Carol. The reason for this revocation could be that Alice got hold of some secret she wants to share with Carol, but not with Bob.

The question now is: can we prevent that Bob gets direct accesss to Carol, – and thereby irrevocable authority to invoke Carol – and if so, who's cooperation is needed to prevent this, and what other authority propagation should these behavior restrictions prevent?

It will be a touchstone for the formal systems we describe in this paper, to see how well they can express the necessary behavior restrictions Alice, Carol, and CT have to respect. Since we aim for a *practical* formalism, an important criterion will be how well the modeled abstraction resembles actual code. This resemblance is crucial because we want to provide (semi-automated) support for safely and accurately modeling the authority propagation that is going on at runtime, using static analysis and abstract interpretation.

4 Authority Reduction in Take-Grant Systems

In this section we introduce a slightly modified form of the Take-Grant systems of [BS79] that is better adapted to our presentation but that retains the properties of the original formalism. This formalism is the basis from which we will evolve Authority Reduction Systems in Sections 5, 6, and 7.

Take-Grant systems are configurations of subjects propagating information and capabilities, represented in a directed graph of nodes with labeled arcs. Rights are represented by labels on the arcs in the graph. Capabilities are represented by these labeled arcs. The nodes (subjects) represent entities that can use capabilities (outgoing arcs) and to which rights can be applied(incoming arcs) via capabilities.

Some entities will not use their capabilities to propagate information and capabilities. These are called "objects" in the original paper [BS79], but we will refer to them as "passive subjects". We will indicate the subjects that need to be active by a bold circle.

Capabilities can have (any combination of) four rights:

`Take` enables a user (Alice) to "take" any capability from the subject (Bob) at the end of the Take-arc. Graphically it means that (a subset of) the arcs originating at Bob are duplicated and given an origin at Alice, and labeled with a subset of the original rights. (Figure 2 left)

Fig. 2. Rights Propagation via `Take` (left) and `Grant` (right)

`Grant` enables a user (Bob) to propagate any capability it has, towards the subject (Alice) at the end of the Grant-arc. (including the very Grant capability to Alice, used by Bob). Graphically it means that (a subset of) the arcs originating at Bob are duplicated and are given an origin at Alice, and labeled with a subset of the original rights. (Figure 2 right)

`Read` enables a user to read information from the subject at the end of the Read-arc. (Figure 3 left)

`Write` enables a user to write any information it has, to the subject at the end of the Write-arc. (Figure 3 right)

Fig. 3. Data propagation resulting in De-Facto authority

Whereas `Take` and `Grant` enable the propagation of capabilities (Figure 2), `Read` and `Write` enable the propagation of information (Figure 3). The arrows in bold indicate which capabilities are used to propagate authority or data. A dashed arrow indicates a new capability that became available through propagation by using the bold-arrow capabilities. Dotted arrows labeled `R` represent the closure of information propagation in the graph.

The dotted arcs are labeled with *de facto authority*, as opposed to the labels of normal arcs that indicate *de jure authority* (by right). De facto read authority (`R`) can always be replaced by de facto write authority (`W`) in the reverse direction (not shown in Figure 3).

Remark that passive subjects can enable the propagation of both data and authority by allowing active subjects to use them as a communication channel. Therefore, when two subgraphs should be authority-separated (kept from influencing each other), they can be connected only via paths that have at least two consecutive passive subjects.

Besides taking, granting, reading and writing, an active subject can also:

Create new subjects that initially have no capabilities, and to whom the parent can have all capabilities.

Drop its capabilities totally or partially. When the last capability towards a subject is dropped, the arc itself is removed. 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 (limits of propagation) there is no need to consider the possible dropping of rights or capabilities.

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 (one newly created subject for every subject in the initial graph) is enough to enable maximum propagation of capabilities and data.

4.1 Discussion

The Caretaker Touchstone Let us investigate the expressive power of Take-Grant systems when we model the caretaker pattern of Section 3. It turns out that, to make sure that Bob cannot get a capability to Carol, we must *not* give him either a take- or a grant-capability to CT. A take-capability would immediately result in Bob taking all capabilities from CT. Since CT has to us his capabilities to Carol, it can only be an active subject. If Bob can grant CT access to Bob himself, CT will inevitably grant Carol to Bob. CT's active but restricted behavior as a proxy cannot accurately be expressed.

The Caretaker pattern, when modeled *directly and in a straightforward way* in the Take-Grant formalism, can only be used to provide revocable *read/write* authority to Bob. Can we find a solution by modeling CT as a subgraph of subjects, some of them being passive? This is possible, at the cost of losing any resemblance to a simple implementation of a proxy.

Authority Reduction Only passive subjects can model authority reduction. They model programmed entities that do not (or cannot) exert *any* of their rights. A subject that actively uses its read rights but only passively assists in

the propagation of its take and grant capabilities can only be modeled as an active subject, a safe but generally too coarse approximation.

On the other hand, passive subjects are often too transparent for authority and allow active subjects to use them as a capability channel. A file reference in a capability secure language only provides authority to store and retrieve *data*, not capabilities. Modeling it as a passive subject will be as if it could also store authority, again a very coarse approach. Passive subjects are well fit to model state that is shared between active subjects, but that is not generally useful in capability languages that support some form of concurrency: the practice of (secure) concurrent programming strongly deprecates the use of shared state concurrency [SV05a,VH04,Rei03].

Conceptually the most important drawback of Take-Grant systems to model object-capabilities is their inability to model the dynamics of collaboration. A "Take" right represents static and eternal authority to acquire all capabilities from a the subject the right points to. The behavior of an entity that always refrains from using a certain capability cannot be modeled by simply removing that capability, because then the model would ignore the fact that the unused capability could still be propagated and be used by another subject. In a capability secure program, entities have to collaborate to propagate data and authority. The invoking entity can offer or request authority but the invoked entity's behavior will decide when and what authority it will return. The only real "right" available in capability systems is the right to use the capabilities that you have. Taking, granting, reading and writing are the *possible effects* (authority) of exerting that right. The decision to actually exert a right is up to the invoking entity, but the effect of the right exertion is largely decided by the invoked entity.

We are not the first to notice this lack of modeling power, as can be inferred from a comment in one of the original papers on Take-Grant systems. In [BS79] Bishop and Snyder mention that Ruzzo suggested them *to use "two place" rules, i.e. two vertices connected by an edge, that describe the circumstances under which a "token" (corresponding to the information) can be moved along from one vertex to another.* The authors give the idea the benefit of doubt, as it could lead to an alternative way of modeling de facto transfer that has *an appealing technical simplicity*, but do not pursue the question any further.

5 Static Authority Reduction Systems

In this section we propose a way to statically model a safe approximation to an entity's readiness to collaborate with other entities. We find 3 orthogonal dimensions in the role an entity plays is collaboration:

1. invoking or being invoked
2. emitting or collecting
3. propagating capabilities or data

For a collaboration between two subjects to succeed, one subject should invoke the other, either one should emit and the other should collect, and both should

be compatible in their modus of propagation (data or capabilities). We assume that all entities differentiate data from capabilities.

The possibility of invoking will be indicated with the prefix `i`, the possibility of responding with the prefix `r`. Emitting capabilities will be indicated with `G` (grant), emitting data with `W` (write). Collecting capabilities will be indicated with `T` (take), collecting data with `R` (read). This gives us eight independent properties of an entity's collaborative behavior to model, as presented in table 1

Table 1. Eight independent aspects of subject behavior

		capabilities	data
invoker	collecting	`iT`	`iR`
	emitting	`iG`	`iW`
responder	collecting	`rT`	`rR`
	emitting	`rG`	`rW`

Instead of considering four rights to define the type of authority a capability carries, we will now have only one: *access*. This is the irrevocable and eternal right to invoke the entity designated by the capability. Access is also a necessary condition for emitting: one can only emit what one has access to. The potential authority carried by a capability depends on the collaborative behavior of the entity designated by it. The actual authority also depends on the collaborative behavior of the owner of the capability.

Table 2. Take-Grant authority from collaboration

Invoker	Responder	Actual Authority	Comments
`iT`	`rG`	`Take`	Invoker might try to collect capabilities, responder might emit them when being invoked
`iG`	`rT`	`Grant`	Invoker might try to emit capabilities, responder might collect them when being invoked
`iR`	`rW`	`Read`	Invoker might try to collect data, responder might emit data when being invoked
`iW`	`rR`	`Write`	Invoker might try to emit data, responder might collect data when being invoked

The new model can also be seen as Take-Grant systems in which the static "rights" are replaced by static authority. Instead of propagating different kinds of rights, only *access* is propagated directly. Indirectly, authority can be propagated too, because the propagation of access will usually give rise to new authority. The authority is static in the sense that the behavior of each of the two collaborating subjects is a static approximation of the behavior of the respective entities they

model. Table 2 shows in what case the authority that corresponds to the four former rights can be generated by collaboration, given that the invoker has access to the responder.

Just like subjects in Take-Grant systems can drop their rights and capabilities, a subject will be able to drop access to another subject in Static Authority Reduction Systems. However, just like in Take-Grant systems, this possibility will not be taken into account when calculating safety properties. A future extension, shortly discussed in Section 9, will allow us to explicitly model dropping access, for reasons of additional expressive power.

Subject Creation: Parenthood and Endowment Like in Take-Grant systems, a subject can only be created by a parent. By *parenthood*, the parent will be the only subject that has access to its child right after the creation. Because the child's cooperation is needed for further propagation of access to it, we have to model *endowment* explicitly. A child could very well be unwilling to accept any authority its clients want to grant to it, but upon creation the parent can always "impose" part of its access to the child. In that way, endowment is not very different from imposing initial conditions on an access graph of communicating entities.

This "imposing" of information and authority is only possible by endowment, and can be an indication to the child that its authority came from its parent. However, this kind of discriminative knowledge will become useful only in Section 7, when subjects will be able to decide their behavior based on what they can observe from their environment.

5.1 Authority Reduction

The eight independent collaborative aspects of behavior allow us to model entities as one of 2^8 different types of subjects. Table 3 shows 15 of these 256 subject types, including both that were available in Take-Grant Systems.

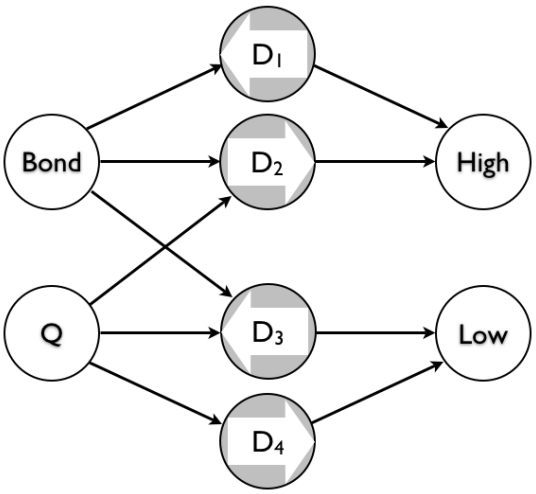
Take-Grant systems have only two types of subjects but they have four independent rights and therefore $2 \times 2^4 = 32$ types of capabilities. We can therefore claim a gain in expressive power by a factor $256 \div 32 = 8$ compared to Take-Grant systems. More importantly, by modeling collaboration explicitly, we have a substantial gain in practical applicability to model capability oriented code.

A simple example that illustrates the need for this expressive power is presented in [MS03], where the authors explain that capabilities obviously *can* implement the $*$-property, contrary to the claim made by Boebert [Boe84] in 1984 and supported by Kain and Landwehr [KL87] in 1987. The authors have to resort to an example in E-language source code to make their claim. The data diodes from table 3 can easily be used to model this code in a Static Authority Reduction configuration, and to prove their claim on an abstract level. (Figure 4)

The $*$-property stipulates that high confidential information should not leak to low clearance subjects. Bond in the example (Figure 4) is a high clearance

Table 3. Subject types as subsets of behavior aspects: some examples

Behavior aspects	Take-Grant subject type	Used to model
{iT,iG,iR,iW, rT,rG,rR,rW}	active	Unknown behavior, approximated by maximal collaboration
{rT,rG,rR,rW}	passive	Non-invoking behavior (shared store) approximated by maximal responder behavior
{iT,iG,iR,iW}	_	Non-invokable behavior, approximated by maximal invoker behavior
{iT,iG,rT,rG}	_	Data opacity (capability filter)
{iR,iW,rR,rW}	_	Capability opacity (data filter)
{rT,rR,iG,iW}	_	Broadcaster in forward direction
{iT,iR,rG,rW}	_	Broadcaster in backward direction
{iG,iW,rG,rW}	_	Source (emits only)
{iT,iR,rT,rR}	_	Drain (collects only)
{rR,iW}	_	Data diode in forward direction
{iR,rW}	_	Data diode in backward direction
{rR,rW}	_	Data store (file, shared data-only store)
{rW}	_	Data source (read only file)
{rR}	_	Data drain (write only file)
{}	_	No Behavior (unforgeable token, name)



Subjects	Type	Approximated by aspects
Bond, Q, High, Low	unknown	{iT, iG, iR, iW, rT, rG, rR, rW}
D2, D4	data diode forward	{rR,iW}
D1, D2	data diode backward	{iR,rW}

Fig. 4. The $*$-property expressed with Static Authority Reduction

subject and should have read/write authority to the high confidentiality subject High, but read-only authority to low confidentiality subject Low: *no writing down*. For low clearance subjects (Q in the example) the policy means *no reading up*. Q should have read/write authority to Low but write-only authority to High. Since the behavior of Bond, Q, High, and Low is unknown, it is safe to assume that they are potential conspirators trying to break the policy.

The fact that this policy holds in the configuration of Figure 4 can easily be derived from the restrictions on de data diodes D_1 through D_4. First of all, no capabilities can be propagated as all connections are to/from data filtering devices. Q has 3 capabilities, of which only D_3 returns information. Given the direction of this diode, this information can only come from Low. Only D_4 can give information to Low, and D_4 can only get its information from Q. This closes the circle: only Q and Low can influence Q and Low.

5.2 Precautions when Modeling Behavior

The proof is nice but there is another important reason why the formal approach is preferable. All assumptions are made explicit now, and we can deduce the requirements for the real code that implements D_1 through D_4. Most important: D_2 and D_3 should be carefully implemented to avoid that they enable one of their clients (Bond) to signal to the other (Q). If Q can observe Bond's usage of D_2 or D_3, then Bond can modulate that usage to signal to Q. The formal approach reveals that special care should be taken when implementing D_2 and D_3, whereas D_1 or D_4 need no such special attention.

The fact that some extra analysis is needed here stems from the fact that the model only expresses knowledge about what a subject will *not* do. The aspects {`rR,iW`} of the forwarding data diodes mean they *could possibly* forward data, but *certainly do nothing else*. This negative knowledge is necessary to calculate safety properties. But when the actual behavior of a subject is observable, the uncertainty about the actual behavior in the model can hide an overt data communication channel.

Every subject that is invokable (modeled with at least one `r`-prefixed aspect) should also carry the {`rW rR`} aspects, unless extra precautions are taken not to be influenced by invocations (`rR`) and not to leak information during invocation (`rW`). Throwing exceptions is but one obvious example of observable behavior that should not be overlooked when modeling and entity's behavior as a set of collaborative behavior aspects.

Conversely, subjects with at least one `i`-prefixed aspect, modeling uncertainty about an entity's invoking behavior, should be augmented with {`iW iR`}, unless extra precautions are taken not to influence the responder (`iW`) and not to be influenced by what can be learned from (trying) actual invocations (`iR`).

The precaution is even more important in Take-Grant systems. Because they use explicit take-grant rights to model what *could* happen, every right *from* an active subject should be accompanied by `Read` and `Write` rights, and so should every right *to* a passive subject. This observation further diminishes the practical use of Take-Grant systems for actual safety analysis.

5.3 Saturation

Until now, we considered the propagation of authority and data via collaboration, but not yet via parenthood and endowment. When investigating propagation by parenthood and endowment, we realized that it has no influence on the safety properties, if some simple conditions are met.

Definition (Saturation) We call a configuration *saturated* when parenthood and endowment do *not* lead to extra propagation of authority or data amongst the subjects in the original access graph. Definition 6 in the appendix provides the formal definition of saturation.

Theorem The maximal propagation of authority and access is not influenced by the effects of parenthood and endowment, if the following conditions are met:

1. The parent has access to itself before creating the child.
2. The child's behavior aspects are a subset of the parent's behavior aspects.

The proof of this theorem is by induction on the propagation steps, and is provided in an appendix (Theorem 1).

Take for example an *unknown* subject that has access to itself. By creating maximally endowed offspring subjects with maximal collaborative behavior (also of type *unknown*), it can introduce extra paths in the access graph, to propagate authority and data. These extra paths cannot add to the propagation of authority amongst the original subjects however, because for every propagation from one original subject to another one, that involves the offspring, there will be an equally valid propagation path that involves only the parent subject.

Corollary This result allows us to model subject creation *implicitly*, by modeling entities that can create offspring as follows:

1. give the subject modeling the parent entity access to itself
2. add to the subject's behavior aspects the union of the behavior aspects of all its possible offspring (for all generations).

This is a very practical way of approximating the unbounded creation of new entities. In some pure object oriented languages every object can have access to itself by default anyway.

The above theorem has a nice consequence for unknown (untrusted) subjects. By giving all unknown subjects self-access, the effects of subject creation by unknown subjects on the propagation of authority and data is completely taken into account.

When the analysis of data and authority propagation is still too coarse grained, the last resort is to start from a more elaborate initial configuration, that includes some of the created subjects.

5.4 The Caretaker Touchstone

As shown in Section 5.1, Static Authority Reduction Systems can model a simple implementation of the ∗-property better than Take-Grant systems, but what about our comparison standard, the caretaker pattern? For ease of reasoning, Figure 5 shows the caretaker pattern in a simplified version.

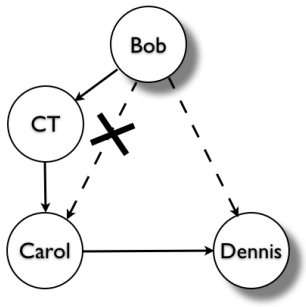


Fig. 5. The caretaker pattern: a simplified version of Figure 1

Again, there is no way that Bob can get direct access to Dennis without also getting direct access to Carol. To understand what is wrong, let us first consider CT's behavior aspects that would allow him to *emit* Dennis to Bob, and see if one of them is safe:

`rG:` When emitting as a responder, CT would immediately emit Carol to Bob.
`iG:` When emitting as an invoker, CT should not get access to Bob, because then he would actively grant Carol to Bob.

This means that CT cannot *emit* Dennis to Bob without also emitting Carol to Bob. Since CT is not able to emit Dennis to Bob in a safe way, Carol has to do the emitting of Dennis to Bob. To enable that, CT should propagate Bob to Carol, and that is impossible without CT itself getting access to Bob. Therefore CT needs `rT` behavior, allowing Bob to grant himself to CT. Then Carol should "take" Bob from CT, but that is impossible because CT would need `rG` behavior to enable this collaboration (see above).

Although we have gained some necessary expressiveness compared to Take-Grant systems (Section 5.1), it would seem we still need more. In Static Authority Reduction Systems as presented here, a subject can neither:

– differentiate its behavior towards different subjects it has access to, nor

– differentiate its behavior towards subjects it has collected actively (as invoker) vs. passively (as responder).

Section 7 will provide a solution for both problems, but we will first investigate the problem of differentiating behavior in Section 6, and see how far we can get with that.

6 Extensions for More Expressiveness

The research work for this section directly followed from our frustrating inability to model a suitable forwarding CT object in the caretaker pattern. We approach the problem pragmatically and introduce an extra set of behavior aspects, to express the fact that a subject is willing to propagate a capability without necessarily also emitting to or collecting from the subject designated by that capability.

We thus add the possibility to express that an entity only passes capabilities, without using them, either forward (collecting as a responder, and emitting as an invoker) or backward (collecting as an invoker, and emitting as a responder).

The rationale is simple and sound: it was the data diodes that allowed us to model the $*$-property in a practical way. We will also have *capability* diodes by modeling the *will-pass-but-not-use* behavior towards collected capabilities. This corresponds directly to the possible behavior of a pure proxy entity like CT in the caretaker pattern.

The extra behavior aspects are presented with a $\sim$ prefix, to indicate:

$\sim$`G` : collecting capabilities as a responder, with the sole intent to emit them as an invoker. (Forward capability diode)

$\sim$`T` : collecting capabilities as an invoker, with the sole intent to emit them as a responder. (Backward capability diode)

$\sim$`W` : collecting data as a responder, with the sole intent to emit it as an invoker. (Forward data diode)

$\sim$`R` : collecting data as an invoker, with the sole intent to emit it as a responder. (Backward data diode)

Equipped with these extra aspects, we show some new and useful types of subjects in Table 4. Observe that the extended aspects are no longer completely orthogonal to the standard ones: $\sim$`G` is automatically implied by subjects that already have the aspects {`rT,iG`}, $\sim$`T` is automatically implied by subjects that already have the aspects {`iT,rG`}, and similar for $\sim$`R` and $\sim$`W`.

The converse is not true however: neither for capability diodes nor for data diodes. The difference is in what will happen to data (authority) that was not collected, but available from initial conditions or endowment. The standard data diodes from table 3 will emit this data, while the ones from table 4 will not (unless of course the same data is re-collected).

Table 4. Subject types with extended behavior aspects: some examples

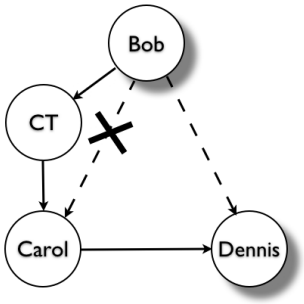
Aspects	Implied	Used to model
{iT,iG,iR,iW, rT,rG,rR,rW}	{∼T,∼G,∼R,∼W}	Unknown behavior, approximated by maximal collaboration
{iT,iG,rT,rG, ∼R,∼W}	{∼T,∼G}	Secret data keeper, will not reveal any data provided by initial conditions or endowment
{iR,iW,rR,rW, ∼T,∼G}	{∼R,∼W}	Authority keeper, will not reveal any capability provided by initial conditions or endowment
{iR,iW,rR,rW}	{∼R,∼W}	Capability opacity (data filter)
{∼R,∼W}	_	Data relay (direction sensitive two-way data filter)
{rT,rR,iG,iW}	{∼G,∼W}	Forward broadcaster, will forward to all capabilities is collects
{∼G,∼W}	_	Forward diode, will forward only to capabilities acquired via initial conditions or endowment
{iT,iR,rG,rW}	{∼T,∼R}	Standard backward broadcaster, reveals its initial capabilities and data
{∼T,∼R}	_	Backward broadcaster, does not reveal its initial capabilities or data
{rR,iW}	{∼W}	Standard data diode in forward direction
{∼W}		Data diode in forward direction
{iR,rW}	{∼R}	Standard data diode in backward direction
{∼R}		Data diode in backward direction

Conclusions: Experiments with these extensions revealed two major results:

1. **Caretaker** We can now model a working caretaker formalism without resorting to composed subjects. Figure 6 shows the caretaker configuration with the behavior restrictions of Carol and CT in the table at the bottom. Alice is out of the picture, but she can have all behavior aspects except iG, to make sure that she does not introduce Carol to Bob herself. All subjects are assumed to have self-access. Carol will possibly emit Dennis but never herself, because she only submits what she can collect. The possibility that Bob can get access Dennis is therefore not disabled, while the possibility that Bob can get access to Carol is.

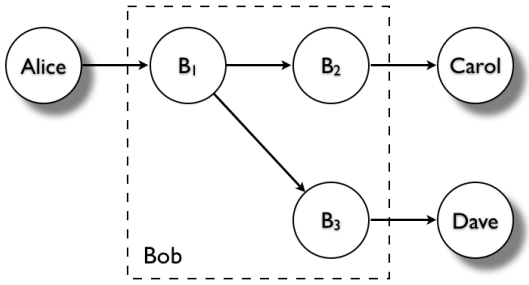
2. **Differentiation via Composite Systems** When complex subjects are composed from simple subjects, the composed subjects can now differentiate their behavior towards different subjects they have access to. This is illustrated in Figure 7.

The figure shows a composite subject Bob that differentiates its influence propagation policy towards the two distinct subjects it has access to: Carol and Dave. If Alice grants a capability, Bob will make sure that it is passed to Carol, but not to Dave. Alice is allowed to write information to Dave, but not



Subjects	Aspects	Comment
Bob and Dennis	{iT,iG,iR,iW,rT,rG,rR,rW	Unknown behavior
CT	{∼T,∼G,∼R,∼W}	Direction sensitive 2-way relay
Carol	{iT,rT,∼T,∼G,iR,rR,iW,rW}	Do not emit what is not collected.

Fig. 6. Behavior of the subjects in the caretaker



Subjects	Aspects	Comments
Alice, Carol, Dave	{iT,iG,iR,iW, rT,rG,rR,rW}	Unknown behavior
B_1	{∼G,∼R,∼W}	Capability diode forward in parallel with Direction sensitive 2-way data relay
B_2	{∼G,iR,rW}	Capability diode forward in parallel with standard backward data diode
B_3	{iT,iW,rR}	Backward capability drain in parallel with standard forward data diode

Fig. 7. Differentiating composite behavior using extended aspects

to pass capabilities to him. Here is an overview of what kinds of things Bob can guarantee, on condition that Dave is not connected via any other paths to Alice, Bob, or Carol. The reader is encouraged to verify the statements.

- Dave will never have access to Alice, Bob (any part of Bob), or Carol
- Carol and Alice will never have direct access to Dave, but they might influence him.
- Carol can only have access to Bob if Alice cooperates. (e.g: Alice grants B_1 to Carol)
- Carol can only influence Dave if Alice cooperates. (e.g.: Alice grants herself to Carol and then passes all influence from Carol via Bob to Dave). Without Alice's cooperation, Bob will not leak information from Carol to Dave.
- Dave can influence (a part of) Bob (B_3) by allowing Bob to take access to whatever subject Dave might have access to, but this access will stay confined to Bob's B_3 part, and never influence Alice or Carol. Bob will treat all subjects he takes from Dave the same way as he treats Dave: accessed via B_3.

The possibility to let composite subjects model abstractions in programming constructs is explained in [MS03]. Composite Authority Reduction Systems are an important step towards the formalization of that idea.

7 Conditional Collaboration

When modeling the propagation of authority and influence, we are now able to express twelve aspects of a subject's behavior. Without resorting to composite subjects, we still cannot model a restriction in Alice's behavior towards Bob without at the same time restricting her possibilities to collaborate with Alice. For Alice, the fact that she cannot return herself when being revoked, means that she also cannot return Dennis or any other capability she might have accepted from the Caretaker. Because attempts to solve this problem in Section 6 complicated the modeling by introducing more aspects, some of which were no longer independent, we now focus on modeling conditional behavior.

Knowledge A subject can (only) base its behavior on knowledge about its environment. This knowledge can be built into the subject's behavior (default, endowed knowledge), or it can become available through collaboration (deduction from experience). The availability of knowledge that can *cause* collaboration should never be underestimated and should be approximated from above. For safe approximation we need not directly model knowledge that can *prevent* collaboration, just as we do not need to consider subjects deliberately dropping access. Non-collaboration conditions should not be overestimated and are therefore only expressed by the absence of corresponding collaboration conditions.

Knowledge is positive and eternal in our model, but it can grow. A subject can react to changes in its environment by learning more, not by forgetting previous knowledge. Conditional collaboration will wait (possibly forever) until the subject can know the truth of the condition, before being activated.

Monotonic Evolution We now have three monotonically changing conditions, depicted in Figure 8. New knowledge can increase a subject's readiness to collaborate and successful collaboration (propagation of information and access) can provide new access to data and capabilities, and can provide new knowledge about the subject's environment.

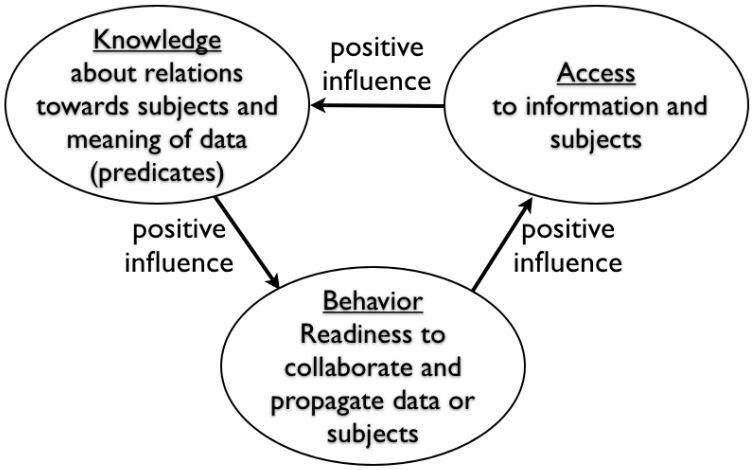


Fig. 8. A subject's monotonic view on the world

By forcing monotonic evolution of knowledge and access we inherit the rich semantics of Concurrent Constraint Programming (CPP) [Sar93] and the expressiveness that comes with it. A conditional collaboration is an *ask* operation, waiting in its own thread until the condition is either entailed or disentailed by the constraint store. We also use CCP to implement a model checker that simply mimics the monotonic evolution of knowledge by *telling* initial conditions and subject behavior to the store, and waiting until the store has reached a global fix-point.

Because we want this to be feasible in a finite amount of time, to keep the system tractable, we allow only a finite number of initial conditions, a finite quantity of discrete knowledge, and a finite upper limit to propagation of access and data in any configuration. This is not a real restriction, in fact it brings an important advantage when controlling the precision of the model: not only can infinite precision be arbitrarily approximated with unbounded finite precision, but this refinement can be applied selectively to specific areas in the model.

Returning to the caretaker pattern, we can now express that Alice could give Bob access to the Caretaker but not to anything else. Alice's "emitting-invoking"

behavior will read: I will invoke X and emit Y to it if X is Bob and Y is the caretaker. This rule states a *sufficient* condition for Alice to grant Y to X. The "but to nothing else" part is modeled by the *absence* of any rule that states that another condition can enable Alice to emit something to Bob.

A subject's knowledge about its environment will be expressed as a set of n-ary predicates representing relations between the subject and a tuple of subjects and/or data. Subject behavior is a set of universally quantified implications (Horn clauses), the condition of which (the body of the Horn clause) is a conjunction of predicates representing knowledge.

Behavioral aspects are expressed as predicates too and are listed in Table 5. The arguments of the predicates show what can be decided by collaborating subjects in a capability system: the invoker chooses what subject it invokes (iGrant, iTake) but the emitter chooses what item it emits (iGrant, rTake). Moreover the invoked subject has no knowledge about who the invoker is (rGrant, rTake), and the collector does not know what subject it will collect (iTake, rGrant).

Table 5. Subject Behavior Aspects as Predicates

predicate	arity	comments
iGrant(S, X)	2	will possibly invoke subject S and emit item X to it
iTake(S)	1	will possibly invoke subject S and collect from it
rGrant()	0	will possibly collect from invokers
rTake(X)	1	will possibly grant item X to invokers

The effects of successful collaboration will become available to the collaborating subjects as read-only predicates, listed in Table 6. Notice that the propagated item (the parameter of the invocation) becomes known to the collector (iTook, rGranted), but the invoker (S) remains unknown to the invoked subject.

Table 6. Effects of Collaboration as Predicates

predicate	arity	comments
iGranted(S, X)	2	succeeded in invoking subject S and emitting item X to it
iTook(S, X)	2	succeeded in invoking subject S and collecting item X from it
rGranted(X)	1	succeeded in being invoked and collecting item X from the invoker
rTaken(X)	1	succeeded in being invoked and emitting item X to the invoker

Besides the predefined predicates listed in Tables 5 and 6, subjects can use other predicates to express their knowledge. As an example, Table 7 shows the Caretaker's behavior as a set of Horn clauses. When the condition is always true it is not shown and the Horn clause is shortened to a fact. Predefined predicates are put in bold, variables are in uppercase and constants are in lowercase.

Table 7. The Caretaker's Proxy Behavior as a set of Horn Clauses

iGrant(S, X)	:- proxyFor(S) ∧ relayForward(X)
iTake(S)	:- proxyFor(S)
rGrant()	
rTake(X)	:- relayBackward(X)
relayForward(X)	:- **rGranted**(X)
relayBackward(X)	:- **iTook**(_, X)
proxyFor(carol)	

The Caretaker is a proxy and will invoke only the subject(s) it is a proxy for (iGrant, iTake). Items granted to the Caretaker by its invokers will be accepted and forwarded to the proxy (rGrant, iGrant, relayForward). To invokers that collect items from it, the Caretaker will emit the items it can collect from (invoking) its proxy (iTake, rTake, relayBackward). Upon initialization, the Caretaker will be told that it is a proxy for Carol. In the definition of relayBackward, the subject is not relevant and replaced by "_" . Note that in this simple case, the use of relayForward(X) and relayBackward(X) could have easily been replaced by their definitions.

Implicit Subject Creation When the behavior of a configuration of subjects does not depend on knowledge about the identity of a subject, the capability rules provide an easy way to take the creation of arbitrary many subjects into account. We are confident that the approach taken in Section 5.3 (safe approximation in saturated systems), can be applied to the dynamic Authority Reductions systems explained in this section. A formal proof of this assumption is future work.

8 Related Work

Another invited talk presented at the TGC'05 workshop by Joshua Guttman, [Jos] showed us how in their work the authors model a dialog between two parties that accumulate monotonically growing knowledge, and we realized that in their setting, the accumulated knowledge can result in less cooperation as well as in more cooperation. A "simplistic" way to model protocols would have involved temporal logic to constrain the order between the events. The fact that they were able to avoid this, gave us hope that there could be a way for us to model a conditional *decrease* in collaborative behavior without resorting to non-monotonic modeling techniques such as (default) timed concurrent constraint programming [SJG95] or temporal concurrent constraint programming[NPV02].

Shortly after we found a solution for this problem. We will simply model two CT subjects: one exhibiting the CT behavior before the revocation and one that will be created as soon as the revocation conditions apply, to exhibit the post-revocation behavior of CT. To model the uncertainty about the actual time of the revocation, the creation of the second one will not disable the first one. In the

future we will simply require that all authority and information between Carol and Bob flows only through the pre-revocation-CT subject. That way we will be able to analyze successive "states" of a subject without loosing monotonicity and tractability in our model.

Another approach to security analysis that is also based on an expressive model for the specifications of security-critical systems can be found in Jan Jürjen's work on Secure Systems Development with UML [JÖ5]. His work extends UML by using the built-in extension features (stereotypes, tagged values, and constraints), for expressing security-related properties. This approach is similar to Authority Reduction Systems in the sense that it also models software (specifications) and then reasons on safety properties in the model. Reasoning in secure UML is done in first order logic.

9 Ongoing and Future Work

The formalized presentation of the Authority Reduction Systems presented in Sections 5 and 6 are presented in the Appendix. We are currently working on the formalization of the concepts introduced in Section 7. In the appendix we provide a proof for the saturation theorem of Section 5. Most of the reasoning in this proof can be applied to the Authority Reduction Systems of Sections 6 and 7 as well. We will give a similar formal proof for these system in a dedicated technical report.

The safety properties is currently express constraints on the the eventual *effects* of authority propagation. It is better to reason about the *flow* of authority. Recent work on graph reachability constraints [QVD05] enables us to impose safety properties expressed as constraints in graphs that are derived from the access graph. The arcs in these derived graphs will indicate the flow of a specific kind of authority or information. We will then be able to express a more precise safety property for the caretaker pattern: "Carol's authority should be *reachable* for Bob only via the caretaker".

To further enhance the expressive power of Authority Reduction Systems, we plan to add the possibility to explicitly and conditionally create new subjects. Because we want the safety properties to remain tractable, we will use the technique of approximation by saturation (Section 5.3) after a given number of creations.

We have build a model checker for the models discussed in SectionstaticArs, and we are currently extending this tool to include the formalism of Section dynamic. The latter tool is not only be useful to check the safety properties in a given configuration of collaborating subjects, but also to calculate the maximal permissive behavior of any subject in the configuration, given a set of global safety properties that have to be guaranteed. Using this tool, we will investigate patterns of capability based collaboration (capability patterns) to discover the limitations of their use. We anticipate that later versions of this tool will also allow us to discover new and useful capability patterns.

Acknowledgments

This work was partially funded by the EVERGROW project in the sixth Framework Programme of the European Union under contract number 001935, and partly by the MILOS project of the Walloon Region of Belgium under convention 114856. We thank Yves Jaradin for reviewing the formal definitions and proofs in the appendix, Raphaël Collet and Yves Jaradin for discussing the formal aspects of the model, and Boriss Mejias for reviewing our drafts. We thank Mark Miller for his assistance on issues of capability-based security and on the flaws in the existing capability formalisms. Only the authors are responsible for any remaining errors.

References

[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.

[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.

[DH65] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. Technical Report MIT/LCS/TR-23, M.I.T. Laboratory for Computer Science, 1965.

[FB96] Jeremy Frank and Matt Bishop. Extending the take-grant protection system, December 1996. Available at: `http://citeseer.ist.psu.edu/frank96extending.html`.

[Har89] Norm Hardy. The confused deputy. *ACM SIGOPS Oper. Syst. Rev*, 22(4):36–38, 1989. `http://www.cap-lore.com/CapTheory/ConfusedDeputy.html`.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.

[HS73] Jerome H.Salzer and Michael D. Schroeder. The protection of information in computer systems. In *Fourth ACM Symposium on Operating System Principles*, March 1973.

[JÖ5] Jan Jürjens. *Secure Systems Development with UML*. Springer, Berlin, June 2005.

[Jos] Joshua D. Guttman and Jonathan C. Herzog and John D. Ramsdell and Brian T. Sniffen. Programming cryptographic protocols. Technical report, The MITRE Corporation. Availalbe at `http://www.ccs.neu.edu/home/guttman/`.

[KL87] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. *IEEE Trans. Softw. Eng.*, 13(2):202–207, 1987.

[LS77] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977.

[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.

[MSC+01] Mark 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`.

[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.

[NPV02] Mogens Nielsen, Catuscia Palamidessi, and Frank D. Valencia. Temporal concurrent constraint programming: denotation, logic and applications. *Nordic J. of Computing*, 9(2):145–188, 2002.

[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.

[Rei03] Stefan Reich. Escape from mutlithreaded hell. concurrency in the language "e", March 2003. Presentation available at: `http://www.drjava.de/e-presentation/html-english/img0.html`.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, Cambridge, MA, 1993.

[SJG95] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Default timed concurrent constraint programming. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 272–285, New York, NY, USA, 1995. ACM Press.

[SMRS04] Fred Spiessens, Mark Miller, Peter Van Roy, and Jonathan Shapiro. Authority Reduction in Protection Systems. Available at: `http://www.info.ucl.ac.be/people/fsp/ARS.pdf`, 2004.

[SV05a] 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.

[SV05b] Fred Spiessens and Peter Van Roy. A practical formal model for safety analysis in Capability-Based systems, 2005. To be published in Lecture Notes in Computer Science (Springer-Verlag). Presentation available at `http://www.info.ucl.ac.be/people/fsp/auredsysfinal.mov`.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.

Appendix

Formal Authority Reduction Systems

In this appendix we give the formal definition of Authority Reduction Systems (ARS) based on transformations of labeled digraphs. The transformation rules will correspond to capability rules for propagation of data and authority, described in Section 2. The formalism is derived from the more general formal systems we described in [SMRS04]. It corresponds to the Static Authority Reduction Systems introduced in Section 5. We will provide a subclass of "Saturated" Authority Reduction Systems, in which the safety problems of finite configuration are decidable, and show how the safety properties in every ARS can be safely approximated by a corresponding ARS in this class. This will proof the claims of Section 5.3.

We will then show how the formal model can be extended with the extra behavior aspects described in Section 6, and how that affects the theorem on saturation, and its proof. The formal model corresponding to Section 7 will be described in future work.

Definitions

Let *Aspects* be the set $\{\texttt{iT}, \texttt{iG}, \texttt{iR}, \texttt{iW}, \texttt{rT}, \texttt{rG}, \texttt{rR}, \texttt{rW}\}$ of behavior aspects.

Definition 1 (Authority Reduction Systems).
An ARS is a tuple $(\mathbf{S}, \mathbf{B}, \mathbf{P})$ *such that:*

- $\mathbf{S}$ *is a countably infinite set of subjects*
- $\mathbf{B}$ *is a behavior function :* $\mathbf{S} \rightarrow 2^{Aspects}$
- $\mathbf{P}$ *is a parenthood function :* $\mathbf{S} \rightarrow \mathbf{S} : \mathbf{P}^n(x) = x \iff \mathbf{P}(x) = x$

Definition 2 (Authority Reduction Configuration ARC).
Let $A = (\mathbf{S}, \mathbf{B}, \mathbf{P})$ *be an ARS.*
An ARC is a tuple (S, E, A) *such that:*

- $S \subseteq \mathbf{S}$ *contains the subjects of the configuration*
- $E \subseteq S \times S$ *represents the access relation between them*

Given an ARC $C = (S, E, A)$*, we will indicate its components:*
$S_C = S$;
$E_C = E$;
$A_C = A$.

Informally, an ARC is an access graph between subjects of an ARS.

Definition 3 ($\vdash$ and $\vdash^*$).
Let $A = (\mathbf{S}, \mathbf{B}, \mathbf{P})$ *be an ARS.*
Let $C_1 = (S_1, E_1, A)$ *be an ARC,*
Let $C_2 = (S_2, E_2, A)$ *be an ARC with* $S_1 \subseteq S_2$
$C_1 \vdash C_2 \iff$ *one of the following conditions applies:*

$create(p, c, \Delta)$ **:** $\exists p, c \in S_2, \exists \Delta \subseteq \{c\} \times S_1$
$S_2 = S_1 \cup \{c\} \wedge P(c) = p$ *(creation of c by p)*
$E_2 = E_1 \cup \{(p, c)\} \cup \Delta$ *(parenthood)*
$\forall (c, x) \in \Delta : (p, x) \in E_1$ *(endowment)*
$grant(x, a, b)$ **:** $S_2 = S_1 \wedge \exists a, b \in S_1 : E_2 = E_1 \cup \{(a, b)\} \wedge \texttt{rT} \in \mathbf{B}(a)$
and $\exists x \in S_1 : \texttt{iG} \in \mathbf{B}(x) \wedge \{(x, a), (x, b)\} \subseteq E_1$ *(x grants b to a)*
$take(a, x, b)$ **:** $S_2 = S_1 \wedge \exists a, b \in S_1 : E_2 = E_1 \cup \{(a, b)\} \wedge \texttt{iT} \in \mathbf{B}(a)$
and $\exists x \in S_1 : \texttt{rG} \in \mathbf{B}(x) \wedge \{(a, x), (x, b)\} \subseteq E_1$ *(a takes b from x)*

From this we derive the following definitions:

$\vdash^*$ **:** *is the reflexive and transitive closure of* $\vdash$
$\vdash^n$ **:** $C \vdash^0 C$ *and* $C \vdash^n C' \iff C \vdash^{n-1} C'' \wedge C'' \vdash C'$

Definition 4 (The predicate couldGetAccess(C,x,y)).
Let $C = (S, E, A)$ *be an ARC, let* $x, y \in S$
$couldGetAccess(C, x, y) \iff \exists C' : C \vdash^* C' \wedge (x, y) \in E_{C'}$

Definition 5 (The predicate couldGetInfo(C,x,y)).
Let $A = (\mathbf{S}, \mathbf{B}, \mathbf{P})$ *be an ARS*
Let $C = (S, E, A)$ *be an ARC, let* $x, y \in S$
$couldGetInfo(C, x, y) \iff$ *one of the following conditions applies:*

1. $x = y$
2. $\texttt{iR} \in \mathbf{B}(x) \wedge \texttt{rW} \in \mathbf{B}(y) \wedge (x, y) \in E_C$ *(x reads from y in C)*
3. $\texttt{rR} \in \mathbf{B}(x) \wedge \texttt{iW} \in \mathbf{B}(y) \wedge (y, x) \in E_C$ *(y writes to x in C)*
4. $\exists C' : C \vdash^* C' \wedge \exists z \in S_{C'} : couldGetInfo(C', z, y) \wedge couldGetInfo(C', x, z)$

Lemma 1 (The effect of adding access to the graph).
Let $A = (\mathbf{S}, \mathbf{B}, \mathbf{P})$ *be an ARS*
Let C *and* C' *be ARCs over* A *with* $E_C \subseteq E_{C'}$ *and* $S_C = S_{C'}$
Let $x, y \in S_C$
$couldGetAccess(C, x, y) \Rightarrow couldGetAccess(C', x, y)$
$couldGetInfo(C, x, y) \Rightarrow couldGetInfo(C', x, y)$

Proof We will prove that every derivation $\vdash^*$ from C is also applicable from C' and that the second derivation results in more (or equal) access in the graph. We prove that every step in the derivation respects these invariants:

- More (or equal) access in the graph and the same behavior of the subjects, does not prevent a step to be applicable

– More (or equal) access in the graph and the same behavior of the subjects before the step, will result in more (or equal) access in the graph and the same behavior of the subjects after the step.

Both requirements follow directly from inspection of the preconditions and the postconditions of the possible steps in Definition 3. Since the initial configuration C has more (or equal) access than C' and the same behavior as C', the lemma follows from the definitions of couldGetAccess() (Definition 4) and couldGetInfo() (Definition 5).

Lemma 2 (The influence of more collaborative behavior).
Let $A = (\mathbf{S}, \mathbf{B}, \mathbf{P})$ *be an ARS*
Let $A^+ = (\mathbf{S}, \mathbf{B}^+, \mathbf{P})$ *be an ARS with* $\forall x \in \mathbf{S} : \mathbf{B}(x) \subseteq \mathbf{B}^+(x)$
Let $C = (S, E, A)$ *and* $C' = (S, E, A^+)$ *be ARCs*
Let $x, y \in S$
$couldGetAccess(C, x, y) \Rightarrow couldGetAccess(C', x, y)$
$couldGetInfo(C, x, y) \Rightarrow couldGetInfo(C', x, y)$

Proof The proof has the same structure as the proof of Lemma 1. We will prove that every derivation $\vdash^*$ from C is also applicable from C' and that the second derivation can result in the same access in the graph, while the subjects in the graph resulting from the second derivation will obviously still have more collaborative behavior. We prove that every step in the derivation respects these invariants:

– More (or equal) behavior and equal access does not prevent a step from being applicable
– More (or equal) behavior and equal access before the step, will result in more (or equal) behavior and equal access after the step.

Both requirements follow directly from inspection of the preconditions and the postconditions of the possible steps in Definition 3. Since C has the same access as C' and more (or equal) behavior than C, the lemma now follows from the definitions of couldGetAccess() (Definition 4) and couldGetInfo() (Definition 5).

Definition 6 (Saturation).
Let $A = (\mathbf{S}, \mathbf{B}, \mathbf{P})$ *be an ARS*
Let $A_0 = (\mathbf{S}, \mathbf{B}, \mathbf{I})$ *be the ARS derived from A by replacing* $\mathbf{P}$ *by the identity function on* $\mathbf{S}$ *(no creation is possible)*
Let $C = (S, E, A)$ *and* $C_0 = (S, E, A_0)$ *be ARCs,*
C is saturated $\iff \forall x, y \in S$ *both the following conditions apply:*

1. $couldGetAccess(C, x, y) \Rightarrow couldGetAccess(C_0, x, y)$
2. $couldGetInfo(C, x, y) \Rightarrow couldGetInfo(C_0, x, y)$

Corollary 1 (Calculating safety properties in a Saturated ARC). *Since by definition, every step in* A_0 *is also a step in A, the converse implications of definition 6 also hold. This means that in a saturated ARC C, the safety properties :*

$$\neg couldGetAccess(C, x, y)$$
$$\neg couldGetInfo(C, x, y)$$

can be calculated without considering creation steps.

Theorem 1 (Safely approximating safety properties via saturation). *This theorem is the formal version of the theorem in Section 5.3.*
Let $A = (\mathbf{S}, \mathbf{B}, \mathbf{P})$ be an ARS
Let $C = (S, E, A)$ be an ARC,
Define the following:

$$\mathbf{B}^+ : \mathbf{S} \to 2^{Aspects} : \mathbf{B}^+(x) = \bigcup_{c \in S : \exists n \in \mathbb{N} : \mathbf{P}^n(c) = x} \mathbf{B}(c) \tag{1}$$

$$E^+ = E \cup \{(x, x) | x \in S \wedge \exists c \in \mathbf{S} \setminus S, n \in \mathbb{N}_0 : \mathbf{P}^n(c) = x\} \tag{2}$$

Let $A^+ = (\mathbf{S}, \mathbf{B}^+, \mathbf{P})$
Let $C^+ = (S, E^+, A^+)$

1. *C^+ is saturated*
2. *The safety properties in C are safely approximated in C^+.*

Proof (part 1) Let P_S be the parenthood function $\mathbf{P}$ of A^+, applied zero or more times, up to the first element in S :

$P_S(x) = P^k(x)$ with $k = \min_{i \in \mathbb{N}, \mathbf{P}^i(x) \in S} i$ if such a k exists,
$P_S(x) = x$ otherwise
From this definition it can be easily deduced that :

$$\forall C, D : C \vdash^* D : P_S(S_D) = S_C \tag{3}$$

$$\forall x \in \mathbf{S} : \mathbf{B}^+(P_S(x)) \supseteq \mathbf{B}^+(x) \tag{4}$$

First we proof that $couldGetAccess(C^+, x, y) \Rightarrow couldGetAccess(C_0^+, x, y)$ with $C_0^+ = (S, E^+, A_0^+)$ and $A_0^+ = (\mathbf{S}, \mathbf{B}^+, \mathbf{I})$.

We will first prove by induction that for any $C^{+\prime}$ such that $C^+ \vdash^* C^{+\prime}$, there exists a $C_0^{+\prime}$ such that $C_0^+ \vdash^* C_0^{+\prime}$ and $\forall x, y \in \mathbf{S} : (x, y) \in E_{C^{+\prime}} \Rightarrow (P_S(x), P_S(y)) \in E_{C_0^{+\prime}}$.

The base case is true because from the definition of C^+ and C_0^+ follows immediately that $E^+ \subseteq E_0^+$.

For the induction case, let's suppose that $C^+ \vdash^* C^{+\prime\prime}$, $C^{+\prime\prime} \vdash C^{+\prime}$ and there exists a $C_0^{+\prime\prime}$ such that $\forall x, y \in \mathbf{S} : (x, y) \in E_{C^{+\prime\prime}} \Rightarrow (P_S(x), P_S(y)) \in E_{C_0^{+\prime\prime}}$.

We conclude by analysis of the $\vdash$ relation between $C^{+\prime\prime}$ and $C^{+\prime}$.

For a step $take(a, x, b)$ (resp. $grant(x, a, b)$), there exists a $C_0^{+\prime}$ derived from $C_0^{+\prime\prime}$ by a step $take(P_S(a), P_S(x), P_S(b))$ (resp. $grant(P_S(x), P_S(a), P_S(b))$). The

access pre-conditions apply because of the induction hypothesis. The behavior conditions apply because of (4). The conclusion follows from Definition 3.

For a creation step $create(p, c, \Delta)$, we can take $C_0^{+\prime} = C_0^{+\prime\prime}$. For the newly created parent-child access $p \rightarrow c$ (parenthood): $P_S(c) = P_S(p) \in S$ and so from (2): $(P_S(p), P_S(c)) \in E_{C^{+\prime}}$ already (and thus also $(P_S(p), P_S(c)) \in E_{C_0^{+\prime}}$ by induction). The preconditions for creating access via endowment indicate that if access $(c, x) \in \Delta$ is created then $(p, x) \in E_{C^{+\prime\prime}}$ should already have been available. From the induction hypothesis, $(P_S(p), P_S(x)) \in E_{C_0^{+\prime\prime}}$ and since $P_S(c) = P_S(p)$ it follows that $(P_S(p), P_S(c)) \in E_{C_0^{+\prime\prime}}$ already.

Now we proof that $couldGetInfo(C^+, x, y) \Rightarrow couldGetInfo(C_0^+, x, y)$ with $C_0^+ = (S, E^+, A_0^+)$ and $A_0^+ = (\mathbf{S}, \mathbf{B}^+, \mathbf{I})$.
$\forall x \in S : P_S(x) = x$; so $\mathtt{iR} \in \mathbf{B}^+(x) \Rightarrow \mathtt{iR} \in \mathbf{B_0^+}(P_S(x))$
The same goes for `iW`, `rR`, and `rW`.

We conclude by induction on the number of intermediate z in the definition of $couldGetConf$ (Definition 5), replacing any subject x by $P_S(x)$, the access conditions are satisfied by the first part of this proof.

Proof (part 2) We have to prove that:
$\neg couldGetAccess(C^+, x, y) \Rightarrow \neg couldGetAccess(C, x, y)$
$\neg couldGetInfo(C+, x, y) \Rightarrow \neg couldGetInfo(C, x, y)$
As $E^+ \supseteq E$, and $\forall x \in \mathbf{S} : \mathbf{B}^+(x) \supseteq \mathbf{B}(x)$, this follows immediately from Lemmas 1 and 2.

Corollary 2 (Tractability). *Safety properties in ARCs with a finite set of subjects can be tractably and safely approximated in a corresponding saturated ARC, by :*

- *giving the subjects that can create offspring self-access, and*
- *adding to their behavior aspects the behavior aspects of all their potential offspring*

Proof It is trivial to show that in a finite ARC, only a finite number of $grant(x, a, b)$-rules and $take(a, x, b)$-rules can be applicable that have an actual effect on the access graph (adding *non pre-existing* access from a to b). The maximum number of possible access arcs in the graphs also being finite, the corollary follows from Theorem 1. Notice that from the monotonicity of these steps, it can even be deduced that actual order of the chosen steps is not influencing the final result (confluency).

Definition 7 (Extensions).
Let $Aspects^{\sim} = Aspects \cup \{\sim\mathtt{T}, \sim\mathtt{G}, \sim\mathtt{R}, \sim\mathtt{W}\}$

Exentended ARS : *is an ARS as defined in Definition 1, except for using* $Aspects^{\sim}$ *instead of* $Aspects$.

Extended ARC *is an ARC as defined in Definition 2, but over an extended ARS*

Extended Step $\vdash^{\sim}$ **:** *a step as defined in Definition 3, with the following additional possibilities:*

grantFar(x, Ω, a, b) **:** $S_2 = S_1, \wedge \exists a, b \in S_1 : E_2 = E_1 \cup \{(a, b)\} \wedge \texttt{rT} \in \mathbf{B}(a)$
and Ω *is a finite series of length* $k \geq 1$ *of elements in* S *such that*
and $\forall i \in \mathbf{N} : 1 \leq i < k : (\Omega_i, \Omega_{i+1}) \in E_1$ *and* $(\Omega_k, a) \in E_1$
and $\forall i \in \mathbf{N} : 1 \leq i \leq k :\sim \texttt{G} \in \mathbf{B}(\Omega_i)$
$\exists x \in S : \{(x, \Omega_1), (x, b)\} \subseteq E_1 \wedge \texttt{iG} \in \mathbf{B}(x)$
(x grants b to a via a series of forwarding relays between x and a)

takeFar(a, Ω, x, b) **:** $S_2 = S_1, \wedge \exists a, b \in S_1 : E_2 = E_1 \cup \{(a, b)\} \wedge \texttt{iT} \in \mathbf{B}(a)$
and Ω *is a finite series of length* $k \geq 1$ *of elements in* S *such that*
and $\forall i \in \mathbf{N} : 1 \leq i < k : (\Omega_i, \Omega_{i+1}) \in E_1$ *and* $(a, \Omega_1) \in E_1$
and $\forall i \in \mathbf{N} : 1 \leq i \leq k :\sim \texttt{T} \in \mathbf{B}(\Omega_i)$
$\exists x \in S : \{(\Omega_k, x), (x, b)\} \subseteq E_1 \wedge \texttt{rG} \in \mathbf{B}(x)$
(x grants b to a via a series of forwarding relays between x and a)

Extended Safety Properties *Extended versions of* $couldGetAccess()$ *and* $couldGetInfo()$:

couldGetAccess(C, x, y) **:** *completely similar to Definition 4*

couldGetInfo(C, x, y) **:** *as defined in Definition 5, but with 2 extra possibilities:*

readFar(x,y) : $\texttt{iR} \in \mathbf{B}(x) \wedge \texttt{rW} \in \mathbf{B}(y)$ *and*
$\exists$ *a finite series* Ω *with length* $k \geq 1$ *of elements in* S *such that :*
$\forall i : 1 \leq i < k : (\Omega_i, \Omega_{i+1}) \in E$
and $\forall i : 1 \leq i \leq k :\sim \texttt{R} \in \mathbf{B}(\Omega_i) and \in E$
and $\{(x, \Omega_1), (\Omega_k, y)\} \subseteq E$
(x reads from y in C via a series of backwards data relays)

writeFar(x,y) : $\texttt{rR} \in \mathbf{B}(x) \wedge \texttt{iW} \in \mathbf{B}(y)$ *and*
$\exists$ *a finite series* Ω *with length* $k \geq 1$ *of elements in* S *such that :*
$\forall i : 1 \leq i < k : (\Omega_i, \Omega_{i+1}) \in E$
and $\forall i : 1 \leq i \leq k :\sim \texttt{W} \in \mathbf{B}(\Omega_i) and \in E$
and $\{(y, \Omega_1), (\Omega_k, x)\} \subseteq E$
(y writes to x in C via a series of forward data relays)

Extended Saturation : *completely similar to Defintion 6.*

A theorem similar to Theorem 1 for extended ARS and extended saturation can be proven in a similar way. We plan to publish this theorem and its proof in a dedicated technical report.