SCOLL

A Language for Safe Capability Based Collaboration

Yves Jaradin Fred Spiessens Peter Van Roy

Université catholique de Louvain

{yjaradin,fsp,pvr}@info.ucl.ac.be

Abstract

In capability secure systems it is important to understand the restrictive influence programmed entities (e.g. procedures, objects, modules, components) have on the propagation of influence in a program. We explain why Take-Grant systems are not sufficiently expressive for this task, and we provide a new formalism – Authority Reduction systems (AR-systems) – to model collaborative propagation. AR-systems provide safe and tractable approximations of adequate precision for the confinement properties in configurations of collaborating entities.

We propose a domain specific declarative language – SCOLL (Safe COLlaboration Language) – to express the collaborative behavior of subjects, the initial conditions in a configuration, and the requirements about confinement and liveness that are to be ensured. We provide the syntactic structure and an operational and denotational semantics for the language. From experiments with a first implementation, we provide a preliminary result and show how patterns for capability based collaboration can be analyzed and generated.

Keywords language, collaboration, security, safety, capability, pattern, authority, model checking, authority reduction

1. Introduction

In capability secure languages and systems [MSC+01, SV05a, SW00, SDN+04] the mechanism that allows *access rights* to propagate through an access graph of connected program entities (e.g. loaded procedures, functions, objects, agents, components, some having references to others) is essentially the same as the mechanism that enables *data* to flow in the access graph. This mechanism is based on *collaboration* between an invoker entity and an invoked entity. Both entities are programmed with certain behavior that will decide *what* will actually be propagated during a collaboration (invocation), and *in what direction*.

Our domain specific language SCOLL is designed for expressing the propagative behavior-aspects of programmed entities, and for consequently analyzing the *positive* impact of local, entity-specific behavior on the global propagation of influence (authority and information) throughout an evolving access graph. Its practical use is most beneficial in capability systems however, because in these systems *every* critical propagation of access rights is controlled by collaborative behavior.

This allows us to also draw *negative* conclusions from such analysis: if behavior controlled propagation *cannot* provide any access-to-file-F to program entity $Alice$, then $Alice$ *will* be effectively prevented from getting access to file F. The actual mechanisms for behavior-controlled propagation in capability systems are described in Section 2.

The most important propagation mechanism in capability systems involves behavior controlled *collaboration*, in which *both* collaborating entities have control: no propagation will happen unless both entities are programmed to enable it. Relying on restrictively programmed behavior to control propagation, capability systems can allow collaboration among mutually suspicious entities, in all confidence that the global confinement requirements will be respected. Access control policies are *programmed*, relying on the restricted propagative behavior of some of the entities (the ones that are usually called "trusted").

Behavior not only controls the confinement of access, but more importantly the confinement of *authority*. Authority is the whole of effects an entity can potentially cause to the system, by using its access rights. An entity with restricted behavior will only use certain access rights in certain ways, under certain conditions. Thus its behavior will *reduce the authority* of other entities that collaborate with it. For instance, an entity could have direct access to a file, but only use that file to append data to it in a certain format, and thus provide exactly this reduced form of authority to its clients.

In Section 3 we present a new formal system for *authority reduction* in capability systems that can model conditional behavior, and provides a safe, precise, and tractable approximation of authority confinement. This formal system builds on earlier results that had less expressive power [SV05b].

Sections 2 and 3 having described the domain of behavior controlled collaboration and propagation, Section 4 will then present our language for this domain. SCOLL is a very simple declarative language that resembles Datalog, and uses predicates and implications to represent conditional behavior and positive knowledge. The capability rules for propagation are enforced by the language. We describe the language's syntactic structure, and give a complete denotational and operational semantics. The implementation is based on concurrent constraint programming [Sar93], and will be briefly explained too.

We consequently highlight the practical value of using SCOLL in Section 5. Behavior based authority control makes it necessary for the programmer to adhere strictly to the principle of least authority (POLA): program the propagation of authority strictly on a need-to-use base. This is not an easy task, as correctly programming an entity now entangles two opposite concerns: making the entity's behavior permissive enough to help provide the required

[copyright notice will appear here]

global functionality, but restrictive enough to help guarantee the required global confinement.

Programmed abstractions for access control and patterns of collaborative behavior can help, but only if their preconditions and consequences are well understood. We show how SCOLL is used to derive such preconditions, from a partial description of a pattern of capability based collaboration and the constraints representing the global confinement policy. A capability pattern for revocable authority called ”the caretaker” will be examined as an example.

We conclude and summarize what remains to be done in Section 6. Related work is mentioned in Section 7.

2. Capability Patterns

In this section we first introduce the view on capabilities that will be used in the rest of the paper. We also give a brief introduction to the formalism of Take-Grant systems that was designed to analyze capability propagation, and we investigate why it does not suffice. We then explain the requirements for an expressive formal system to be able to safely and precisely analyze patterns of capability based collaboration.

2.1 Capability Based Security

Dennis and Van Horn [DH65] introduced the concept of a capability in 1965. A capability is an unforgeable designation (reference) to a resource that is inextricably combined with an access right to that resource. In capability systems, authority is only available in the form of capabilities, and all references to resources are capabilities. If you are able to reference an entity (via a capability), you are allowed to use it and to pass it on to other entities you have access to. This may seem a very weak and discretional policy at first sight, but a brief explanation will correct that impression.

In [MS03] Miller and Shapiro propose a view on capabilities they call *object-capabilities*. All references to program entities are capabilities that provide the right to *use* (invoke) the designated entity. The authority exerted when using the entity is decided by the entity's programmed behavior. We distinguish the roles entities play in the collaboration as follows:

invoker : the entity that invokes

responder : the entity that is being invoked

emitter : an entity that collaborates by *emitting* (providing) authority or information.

collector : an entity that collaborates by *collecting* (accepting) authority or information.

There is always exactly one invoker and one responder in a collaboration. The invoker decides what entity (or data) will be invoked (among the ones it has access to), the emitter decides what entity will be emitted (among the ones it has access to). To propagate something, one entity has to emit it and the other one has to collect it. These rules for propagation reflect the scoping rules for invocation in an object oriented capability language that provides strict encapsulation [MSC$^+$01, SV05a].

This is the view on capabilities we will use in this paper. It unveils the real preconditions for propagation of authority: entities have to collaborate to pass a capability, and such collaboration involves both entities' behavior. While it is the inalienable and eternal right of the holder of a capability to invoke the designated entity, the effect (authority) that is resorted is dynamic and largely decided (possibly reduced to zero) by the invoked entity's behavior.

It becomes clear in this view that the "discretional" nature of capabilities is not actually a weakness. Propagation is not an eternal right but a potential effect of using a right, that can be restrained by *each* of the collaborating entities. Confinement policies can now be implemented by introducing entities with carefully restricted behavior at strategic places in a configuration (access graph) of mutually distrusting entities.

Besides collaboration, creating new entities can also cause propagation in the following restricted sense:

parenthood : The creating entity (parent) gets access to the created entity (child).

endowment : The parent can endow part of its access to the child, at the time of creation.

These rules reflect the scoping rules for object creation in an object oriented capability language. The parent object assembles the internal state of the child object from the entities the parent can access. In practice this means of course that the parent can set up bidirectional communication channels to its child.

The inextricable combination of and entity's designation with the right to use it has an important advantage over the so called "mandatory" access control policies that separate these concepts. When designation means right-to-use, delegation of authority becomes propagation of capabilities. Deputy entities can now be designed to use the authority provided by their clients in the form of capabilities provided to them by their clients upon invocation, without becoming vulnerable to a confused deputy attack. A confused deputy is an entity that cannot tell the difference between its own authority and the authority that is supposed to be delegated to it by its client. It cannot avoid being lured into using its own authority on its client's behalf, even if this client has no such authority.

As explained in [Har89] and [SV05a], only capabilities can prevent such an attack. Stack-walking mechanisms can help in verifying who delegated what authority for what purpose to whom, and be very expressive and relatively efficient [WBDF97] but still cannot avoid the possibility of confused deputies in general, particularly since they provide no solution for concurrency and distribution.

A brief overview of the advantages and drawbacks of capability security in the view of object-capabilities is provided in [SV05b]. One important drawback is the lack of orthogonality of the concerns *security* and *functionality*. Confinement becomes completely entangled with functionality, programmed together into the same object methods and procedures. In [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.

Related to this drawback is the need for powerful tools that help design and analyze safe patterns of collaboration. Without such a tool it is very hard for an application developer to precisely assess what effects programmed behavior has on the global confinement requirements. Our main contribution in this paper, is to provide a formalism and a language that will be the basis for such a tool.

2.2 Take-Grant Systems

We will now briefly introduce the Take-Grant systems of [BS79]. For a more detailed account on the particularities of the Take-Grant formalism with respect to propagation and collaboration, we refer to [SV05b].

Take-Grant systems are configurations of subjects propagating information and capabilities. The configurations are labeled directed graphs of nodes representing the subjects, arcs representing access, and labels on the arcs representing sets of access-rights to the subject pointed at by the arc. Capabilities are the labeled arcs in a configuration. The subjects (nodes) model entities that can use capabilities (outgoing arcs) and to which rights (incoming arcs) can be applied via capabilities.

Two rights govern the propagation of capabilities: *grant* and *take*. A *grant*-labeled capability allows the holder to emit any capabilities it holds to the subject pointed at by the capability, including its own take and grant capabilities. A *take*-labeled capabi-

lity allows the holder to collect any capabilities hold by the subject pointed at by the *take*-capability. Figure 1 illustrates both mechanisms: the new capabilities arising from propagation are indicated with a dashed arrow.

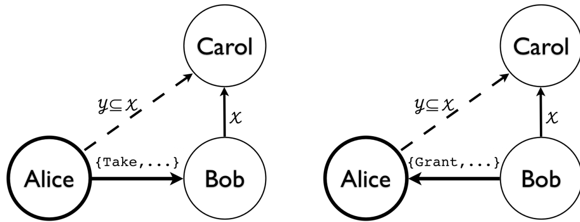

Figure 1. Capability Propagation via `Take` (left) and `Grant` (right)

Some subjects will not use their own rights to propagate information and capabilities. We will refer to them as "passive subjects". Their behavior is restricted in the following sense: passive subjects will only provide and accept capabilities when being invoked.

A subject can also create a new subject and thereby take any rights it wants to it. Subjects can also drop their rights, partially or completely.

The advantages of this formal system are its simplicity and the fact that global confinement properties are tractable [LS77, FB96]. The main drawbacks are its lack of power to express restricted behavior, and the fact that propagation is modeled with rights rather than with authority. The collaborative aspect of propagation is therefor completely lost. As explained in [SV05b], this has added to a certain under appreciation of the fitness of capability systems to guarantee certain forms of confinement [Boe84, KL87].

2.3 Patterns of Collaboration

The formal system we are looking for should allow us to express and analyze useful patterns of safe collaboration. Patterns of safe collaboration are programming idioms for writing capability based secure programs, analogous to object-oriented design patterns [GHJV94]. Such a pattern is useful if its preconditions to be effectively applicable are well understood and described. As an example, let us look at the pattern for revocable authority called *the caretaker* in [MS03].

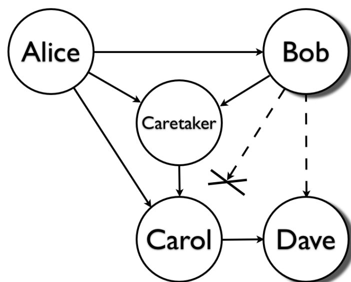

Figure 2. The caretaker pattern for revocable authority

Figure 2 depicts a configuration of collaborating subjects. Shadowed subjects indicate that we make no assumptions about their behavior: these subjects are *unspecified*. In the pattern, Alice wanted to give Bob revocable authority to use Carol, and therefor created a proxy (Caretaker) to Carol and gave that to Bob instead. Alice will also emit what she can collect from collaborations. Carol has access to an unspecified subject Dave. This is the initial configuration depicted by the solid arrows in the figure.

Alice relies on Caretaker's behavior to stop proxying when she sends it a certain message. For this revocation to have effect, Bob should of course never get direct access to Carol (indicated by the arrow ending in a cross). It is OK for Bob to get direct access to Dave though: that is depicted by the dashed arrow from Bob to Dave.

Given the behavior mentioned for Alice and for Caretaker (the latter one being a proxy object to Carol and only *relaying* propagation), it is mentioned in [MS03] that Carol's behavior also has to be restricted in a certain way: she should for instance not return herself when being invoked.

To make the caretaker pattern useful, we have to understand *very precisely* what Carol should not do in this case. In general, given a configuration of collaborating subjects of which the behavior is partially described, and given a set of confinement requirements (what should not happen) and liveness requirements (what should not be prevented from happening), we have to be able to calculate all minimal sets of restrictions in the non-described part of the behavior of any subject in the configuration, that suffice to guarantee these requirements.

3. Authority Reduction Systems

In this section we propose a formalism capable of expressing capability based patterns of collaboration. We will aim for simplicity and expressive power, and devise a formalism that allows to model collaborative behavior at different levels of refinement, while allowing a tractable and safe calculation of confinement.

3.1 Goal

We want our formalism to be practically useful for software engineers during the design and implementation phase:

- to reason about the feasibility of confinement requirements during the design of their program, and
- to verify existing code and check if the confinement requirements are respected.

That means that the necessary translations between existing code and behavior specifications for subjects in the formalism should be straightforward in both directions. To safely model entity code into subject behavior, the collaborative behavior should be approximated from above: making sure that the subject will *collaborate whenever it is not impossible* that the actual entity would collaborate in that fashion. The precision of modeling should be adaptable, so that the model can be refined exactly in the places where it turns out to be too crude.

When refining subject behavior into actual code, it should be easy to interpret the specified restrictions in subject behavior as requirements for the modeled entities.

To achieve the necessary expressive power we model an entity's awareness of its environment as *knowledge* of the subject. A subject's *intent* describes how its *behavior* is positively influenced by its knowledge. This is illustrated in Figure 3. The effects of collaboration will provide more knowledge to the subject. More knowledge can only lead to more collaborative behavior. The mutual interaction between behavior and knowledge in a subject is completely monotonic. The rules that govern collaboration between entities will model the capability rules for propagation of authority and information.

For simplicity we will model neither time nor non-monotonic changes (decrease) in knowledge or collaboration. Section 7 will briefly mention how to model non-monotonic effects in our monotonic system.

To keep confinement tractable, we allow subjects to model sets (aggregates) of entities instead of a single entity. We will investigate and proof the conditions for this to be a safe approximation. Aggregation will be used to implicitly model subject creation (aggregating a entity with all its offspring), but it can also be useful

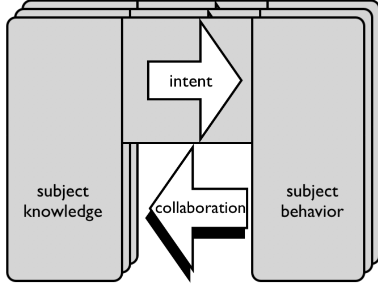

Figure 3. The amplifying influence of knowledge and behavior

to model composite entities (e.g. components), clusters of entities (e.g. unspecified entities that are connected), or entities that have a similar purpose or equal clearance or confidentiality level.

Subject behavior is expressed with the predicates listed in Table 1. The prefix `i` means the subject S_1 invokes, the prefix `r` means it responds to invocation. The lower part of the table shows the subject behavior predicates to create other subjects. No collaboration is needed for creation and endowment: it is subject S_1's sole decision to create subjects and endow it. The prefix `p` means S_1 is the endowing parent, the prefix `c` means S_1 is the child receiving endowment.

Table 1. Predicates for subject behavior

predicate	comments
$iEmit(S_1, S_2, X)$	S_1 tries to invoke subject S_2 and emit X to it
$iCollect(S_1, S_2)$	S_1 tries to invoke subject S_2 and collect from it
$rEmit(S_1, S_2, X)$	S_1 tries to emit X when invoked by S_2
$rCollect(S_1, S_2)$	S_1 tries to collect when invoked by S_2
$create(S_1, S_2)$	S_1 intends to create S_2
$pEndow(S_1, S_2, X)$	Parent S_1 endows its child S_2 with X

These behavior predicates describe the behavior of the subject in its first argument: S_1. The fact that no collaboration is needed for $create()$ and $pEndow()$ does not mean that every creation and endowment will always succeed. It will for instance not be possible for S_1 to endow S_2 without also creating S_2. The rules that govern the actual evolution of a configuration will be defined in Section 3.2.

The $r-$prefixed predicates, corresponding to behavior when being invoked, include the invoker as the second argument. This does not mean that we assume that responders have complete knowledge of their invoker. However, we want to be able to model responder behavior depending on *partial* knowledge about the invoker, available from an invocation. For instance, the fact that an invoker emits X to the responder, might be a precondition for the responder to emit Y to the invoker *in the same invocation*. The language SCOLL will allow this specific type of invoker-dependent behavior, but restrict the use of invoker identity in other cases.

Table 2 lists the knowledge predicates that can be used by a subject to express its intent.

The knowledge provided in these predicates is available *only* to the subject in the first argument (S_1). Notice again that the $r-$prefixed predicates, corresponding to knowledge about being invoked, include the invoker as the second argument. Again, this knowledge will be only used to express knowledge available about an invocation, rather than about the invoker.

Table 2. Predicates for subject knowledge

predicate	comments
$access(S_1, X)$	S_1 has access to X
$iEmitted(S_1, S_2, X)$	S_1 succeeded in invoking S_2 and emitting X to it
$iCollected(S_1, S_2, X)$	S_1 succeeded in invoking S_2 and collecting X from it
$rEmitted(S_1, S_2, X)$	S_1 emitted X to an invoker (S_2)
$rCollected(S_1, S_2, X)$	S_1 collected X from an invoker (S_2)
$cEndowed(S_1, S_2, X)$	child S_1 was endowed with access to X by its parent S_2

Notice also that the c-prefixed predicate $cEndowed$ includes knowledge about the parent, while in general such knowledge is *not* provided in capability systems. To explain why the second argument of $cEndowed()$ nevertheless indicates the parent subject we have to consider a child subject that models an aggregation of entities (e.g. all entities of a certain restricted behavior). Some of the entities modeled by this child subject can be created by different parent entities, that are modeled by different parent subjects. While entities cannot have more than one parent, subjects can. In fact such a subject can even be "created" and endowed by itself. Similar to the invoker in $rEmitted()$ and $rCollected()$, the parent subject in $cEndowed()$ will be used to differentiate a subject's reaction based on what can be derived from the other arguments in the endowment. Thus an aggregate child subject can decide its behavior based on what it is being endowed with by *the same* parent.

3.2 Formal Authority Reduction Systems

In this section we define the basic concepts and constructs for collaborative behavior.

Let $\mathbf{B} = \{\texttt{iEmit}, \texttt{iCollect}, \texttt{rEmit}, \texttt{rCollect}, \texttt{create}, \texttt{pEndow}\}$
Let $\mathbf{K} = \{\texttt{iEmitted}, \texttt{iCollected}, \texttt{rEmitted}, \texttt{rCollected}, \texttt{access}, \texttt{cEndowed}\}$
Let $ar : \mathbf{B} \cup \mathbf{K} \rightarrow \mathbb{N}$ be the arity function of the predicates as defined in Tables 1 and 2.
For an arbitrary set S, define:

$$\mathbf{K}_S = \{k(s_1, \ldots, s_{\mathbf{ar}(k)}) | k \in \mathbf{K}, s_i \in S for 1 \leq i \leq \mathbf{ar}(k)\}$$
$$\mathbf{B}_S = \{b(s_1, \ldots, s_{\mathbf{ar}(b)}) | b \in \mathbf{B}, s_i \in S for 1 \leq i \leq \mathbf{ar}(k)\}$$

DEFINITION 1 (ARS). *Authority Reduction System :*
An ARS is a tuple $(\mathbf{S}, \mathbf{P}, \mathbf{I})$ *such that:*

- $\mathbf{S}$ *is a countable set of subjects, defining:*
- $\mathbf{P} \subseteq \mathbf{S} \times \mathbf{S}$ *is a parenthood relation.*
- $\mathbf{I}$ *is an intent function :* $\mathbf{S} \rightarrow \mathcal{F}(2^{\mathbf{K_S}}, 2^{\mathbf{B_S}})$ *(where* $\mathcal{F}(A, B)$ *is the set of all functions from* A *to* B*) such that:*
  - $\mathbf{I}(s)$ *is only defined for all local knowledge :* $\forall K, s : (\forall k(s_1, \ldots, s_n) \in K : s_1 = s) \iff (\exists B : B = I(s)(K))$
  - $\mathbf{I}(s)$ *generates only local behavior:* $B = \mathbf{I}(s)(K) \implies \forall b(s_1, \ldots, s_{\mathbf{ar}(b)}) \in B : s_1 = s$
  - $\mathbf{I}(s)$ *is monotonic:* $B = \mathbf{I}(s)(K) \land B' = \mathbf{I}(s)(K') \land K \subseteq K' \implies B \subseteq B'$

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

- $S \subseteq \mathbf{S}$ *contains the subjects of the configuration*
- $E \subseteq S \times S$ *represents the access relation between them*
- $K \subseteq \mathbf{K}_S \subseteq \mathbf{K}_{\mathbf{S}}$ *represents the actual (initial) knowledge of the subjects in the configuration.*

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

In the next definition we define the ways an ARC can evolve. The following implications of capability based collaboration are enforced:

parenthood : The parent entity gets access to its child entity upon creation. Since subjects can model sets of entities, the situation can arise that two or more subjects can create a common child subject. The subject will be added to the configuration the first time it is created. Subsequent creation will give the new parent subject access to the child subject.

endowment : The parent entity can give part of its access to its child. The rules make sure that a parent subject can only endow access to subjects the parent has access to.

propagation : The emitter needs access to the subject that is propagated, the invoker needs access to the responder. Emitting data is not explicitly modeled but data can be mimicked by entities with no behavior (no intent). As will be mentioned in Section 6, we have future plans to model data propagation in its own right however, to refine the analysis in situations where data can propagate but capabilities can not.

The following restrictions of capability based collaboration are *not* enforced by evolution in an ARC. They will be enforced by restrictions in Section 4, when the intent functions $\mathbf{I}(s)$ for the subjects will be expressed in the SCOLL language.

endowment : A child does not know the identity of its parent. This principle can be violated if $\mathbf{I}(s)$ expresses behavior based on parent identity.

propagation : A responder does not know the identity of its invoker. This principle can be violated if $\mathbf{I}(s)$ expresses behavior based on invoker identity.

DEFINITION 3 ($\vdash$). *Step*
Let $A = (\mathbf{S}, \mathbf{P}, \mathbf{I})$ be an ARS.
Let $C_1 = (S_1, E_1, K_1, A)$ be an ARC.
Let $C_2 = (S_2, E_2, K_2, A)$ be an ARC with $S_1 \subseteq S_2$
$C_1 \vdash C_2 \iff$ one of the following conditions applies:

$creates(p, c)$ **:** $\exists p, c \in S_2$:
$S_2 = S_1 \cup \{c\}$
$(p, c) \in \mathbf{P}$
$\exists K_p \subseteq K_1 : \mathtt{create}(p, c) \in \mathbf{I}(p)(K_p)$
$K_2 = K_1 \cup \{\mathtt{access}(p, c)\}$
$E_2 = E_1 \cup \{(p, c)\}$

$endows(p, c, x)$ **:** $\exists p, c, x \in S_1$
$S_2 = S_1$
$\{(p, c), (p, x)\} \in E_1$
$(p, c) \in \mathbf{P}$
$\exists K_p \subseteq K_1 : \mathtt{create}(p, c) \in \mathbf{I}(p)(K_p)$
$K_2 = K_1 \cup \{\mathtt{cEndowed}(c, p, x), \mathtt{access}(c, x)\}$
$E_2 = E_1 \cup \{(c, x)\}$

$grants(x, a, b)$ **:** $\exists a, b, x \in S_1 : S_2 = S_1$
$\{(x, a), (x, b)\} \subseteq E_1$
$\exists K_x \subseteq K_1 : \mathtt{iEmit}(x, a, b) \in \mathbf{I}(x)(K_x)$
$\exists K_a \subseteq K_1 : \mathtt{rCollect}(a, x) \in \mathbf{I}(a)(K_a)$
$E_2 = E_1 \cup \{(a, b)\}$
$K_2 = K_1 \cup \{\mathtt{iEmitted}(x, a, b), \mathtt{rCollected}(a, x, b), \mathtt{access}(a, b)\}$

$takes(a, x, b)$ **:** $\exists a, b, x \in S_1 : S_2 = S_1$
$\{(a, x), (x, b)\} \subseteq E_1$
$\exists K_a \subseteq K_1 : \mathtt{iCollect}(a, x) \in \mathbf{I}(a)(K_a)$
$\exists K_x \subseteq K_1 : \mathtt{rEmit}(x, a, b) \in \mathbf{I}(x)(K_x)$
$E_2 = E_1 \cup \{(a, b)\}$
$K_2 = K_1 \cup \{\mathtt{iCollected}(a, x, b), \mathtt{rEmitted}(x, a, b), \mathtt{access}(a, b)\}$

From this we derive the following definitions:

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

(In fact $C \vdash^* D \iff \exists n \in \mathbb{N} : C \vdash^n D$)

The evolution of a configuration via $\vdash^*$ can model *all possibilities* for propagation in a capability system, be it that AR-systems are a bit too expressive. As mentioned before, the intent functions can take knowledge into account that is not available in normal capability systems (about the invoker or the parent), but this surplus in expressive power will be naturally removed by the specification language for intent that will be presented in Section 4.

To analyze confinement properties in an AR-configuration is to analyze the boundaries of propagation that the configuration allows. We will therefor define a propagation predicate $couldGetAccess(C, x, y)$ for every pair of subjects in a configuration, indicating that evolution in the configuration C will allow (not prevent) subject x to get access to subject y.

DEFINITION 4. $couldGetAccess(C, x, y)$
Let C be an ARC with $x, y \in S_C$
$couldGetAccess(C, x, y) \iff \exists C' : C \vdash^* C' \wedge (x, y) \in E_{C'}$

We also define a predicate for the propagation of knowledge.

DEFINITION 5. $couldGetKnowledge(C, k(s_1, \ldots, s_{\mathbf{ar}(k)}))$
Let C be an ARC
$couldGetKnowledge(C, k(s_1, \ldots, s_{\mathbf{ar}(k)})) \iff \exists C' : C \vdash^* C' \wedge k(s_1, \ldots, s_{\mathbf{ar}(k)}) \in K_{C'}$

3.3 Safe Approximation via Aggregation

When calculating confinement results, a strict one-to-one mapping of programmed entities to subjects in an ARS is not always recommended. In principle, entities can keep on creating other entities, and the configuration might never stop evolving, rendering precise calculation of confinement intractable. We have strong indications that, just like the calculation of safety properties in the protection systems described in [HRU76], the calculation of confinement boundaries in Authority Reduction Systems is intractable in general. We have a scheme of a proof – which we will not provide in this paper – for the Turing completeness of these systems, to support this believe.

To ensure that a reasonable approximation to the confinement result is tractable, we propose the technique of aggregation, that was introduced informally in Section 3.1.

We will proof that the following properties of a configuration with aggregated subjects are sufficient to provide a safe approximation of confinement:

- The aggregate subject should collaborate, create, and endow, as soon as one of its constituting subjects does.
- Any subject should collaborate with the aggregate subject as soon as it would collaborate with one of the constituting subjects.

- Any subject should emit (collect) an aggregate subject as soon as it would emit (collect) one of the constituting subjects.
- Any subject should create an aggregate subject as soon as it would create one of the constituting subjects.
- Any subject should endow a child with access to an aggregate subject as soon as it would endow that child with access to one of the constituting subjects.

This set of rules is formalized in the following definition.

DEFINITION 6. *ARS Aggregation*
Let $A = (\mathbf{S}, \mathbf{P}, \mathbf{I})$ be an ARS.
Let g be a surjective application : $\mathbf{S} \to \mathbf{S}'$
$Agg(g)$ is the ARS-aggregation function defined as follows:
$Agg(g)(A) = (\mathbf{S}', \mathbf{P}', \mathbf{I}')$ where

1. *behavior aggregation*
   $B' = \mathbf{I}'(s')(K') \iff \exists B_1, \dots, B_n, s_1, \dots, s_n, K_1, \dots, K_n$ *such that*
   $\forall 1 \le i \le n : B_i = \mathbf{I}(s_i)(K_i)$
   $\forall 1 \le i \le n : g(s_i) = s'$
   $B' = \bigcup_{1 \le i \le n} B'_i$ *where*
   $b(s'_1, \dots, s'_{\mathbf{ar}(b)}) \in B'_i \iff \exists s_1, \dots, s_{\mathbf{ar}(b)} : b(s_1, \dots, s_{\mathbf{ar}(b)}) \in B_i \wedge \forall 1 \le j \le \mathbf{ar}(k) : g(s_j) = s'_j$
   $K' = \bigcup_{1 \le i \le n} K'_i$ *where*
   $k(s'_1, \dots, s'_{\mathbf{ar}(k)}) \in K'_i \iff \exists s_1, \dots, s_{\mathbf{ar}(k)} : k(s_1, \dots, s_{\mathbf{ar}(k)}) \in K_i \wedge \forall 1 \le j \le \mathbf{ar}(k) : g(s_j) = s'_j$
2. *parenthood aggregation*
   $(p', c') \in \mathbf{P}' \iff \exists (p, c) \in \mathbf{P} : g(p) = p' \wedge g(c) = g'$

DEFINITION 7. *ARC Aggregation*
Let $A = (\mathbf{S}, \mathbf{P}, \mathbf{I})$ be an ARS
Let g be a surjective application : $\mathbf{S} \to \mathbf{S}'$
Let C be an ARC in A
$Agg_c(g)$ is the ARC aggregation function defined as follows:
$Agg_c(g)(C) = D$ where

- $A_D = Agg(g)(A)$
- $g(S_C) = S_D$
- $(s'_1, s'_2) \in E_D \iff \exists (s_1, s_2) \in E_C : g(s_1) = s'_1 \wedge g(s_2) = s'_2$
- $k(s'_1, \dots, s'_{\mathbf{ar}(k)}) \in K_D \iff \exists k(s_1, \dots, s_{\mathbf{ar}(k)}) \in K_C$ *such that*
  $\forall i : 1 \le i \le \mathbf{ar}(k) : g(s_i) = s'_i$

Since aggregation is going to be used to turn potentially intractable problems of confinement into practically tractable problems, we have to be sure that the confinement results are valid. Approximation by aggregation can sometimes provide "false positives", indicating possibilities for propagation where there actually are none, but it should never provide "false negatives", indicating confinement (impossibility of propagation) while the non-aggregated system could detect possibilities for propagation.

THEOREM 1. *ARC aggregation keeps confinement properties*
Let C and D be AR configurations such that $Agg_c(C) = D$

1. $\neg couldGetAccess(D, g(x), g(y))$
   $\implies \neg couldGetAccess(C, x, y)$
2. $\neg couldGetKnowledge(D, k(g(s_1), \dots, g(s_{\mathbf{ar}(k)})))$
   $\implies \neg couldGetKnowledge(C, k(s_1, \dots, s_{\mathbf{ar}(k)}))$

Proof

We will proof the converse implication of part **1** of the theorem: $couldGetAccess(C, x, y) \Rightarrow couldGetAccess(D, g(x), g(y))$ with the help of Lemma 2. The proof of part **2** is completely analogue and is not provided.

LEMMA 1. *Commutativity of $\vdash$ and $Agg_c(g)$*
For any ARS $A = (\mathbf{S}, \mathbf{P}, \mathbf{I})$, any ARC C in A and any surjective application $g : \mathbf{S} \to \mathbf{S}'$,
$C \vdash C_1 \implies Agg(g)(C) \vdash Agg(g)(C_1)$.

Proof of Lemma 1

Let $A' = (\mathbf{S}', \mathbf{P}', \mathbf{I}') = Agg(g)(A)$
By Definition 7, $S_C \subseteq S_{C_1} \implies S_{Agg(g)(C)} \subseteq S_{Agg(g)(C_1)}$.
Since $C \vdash C_1$, one of the conditions from Definition 3 has to be true. If the step was a creation, $create(p, c)$:
$(p, c) \in S_{C_1} \implies (p, c) \in S_{Agg_c(g)(C_1)}$ by Definition 7.
$S_{C_1} = S_C \cup \{c\} \implies S_{Agg_c(g)(C_1)} = S_{Agg_c(g)(C)} \cup \{g(c)\}$ by Definition 7.
$(p, c) \in \mathbf{P} \implies (g(p), g(c)) \in \mathbf{P}'$ by Definition 6
$(\exists K_p \subseteq K_C : \mathtt{create}(p, c) \in \mathbf{I}(p)(K_p)) \implies (\exists K_{g(p)} \subseteq K_{Agg_c(g)(C)} : \mathtt{create}(g(p), g(c)) \in \mathbf{I}'(g(p))(K_{g(p)}))$ by Definition 6 and Definition 7
$K_{C_1} = K_C \cup \mathtt{access}(p, c) \implies K_{Agg_c(g)(C_1)} = K_{Agg_c(g)(C)} \cup \mathtt{access}(g(p), g(c))$ by Definition 7.
$E_{C_1} = E_C \cup (p, c) \implies E_{Agg_c(g)(C_1)} = E_{Agg_c(g)(C)} \cup (g(p), g(c))$ by Definition 7.
Thus, $Agg(g)(C) \vdash Agg(g)(C_1)$.

For other kinds of steps, the proof is similarly trivial and derives from Definition 6 and Definition 7.

LEMMA 2. *Commutativity of $\vdash^*$ and $Agg_c(g)$ For any ARS $A = (\mathbf{S}, \mathbf{P}, \mathbf{I})$, any ARC C in A and any surjective application $g : \mathbf{S} \to \mathbf{S}'$,*
$C \vdash^* C_1 \implies Agg(g)(C) \vdash^* Agg(g)(C_1)$.

Proof of Lemma 2

We will prove this lemma by induction on the number of steps. The base case ($C \vdash^0 C \implies Agg_c(g)(C) \vdash^0 Agg_c(g)(C)$) is trivial since the right part of the implication is true by definition of $\vdash^0$.
The induction case
($C \vdash^* C' \vdash C'' \wedge Agg_c(g)(C) \vdash^* Agg_c(g)(C') \implies Agg_c(g)(C) \vdash^* Agg_c(g)(C'')$) is also trivial since $C' \vdash C'' \implies Agg_c(g)(C') \vdash Agg_c(g)(C'')$ by Lemma 1.

Proof of Theorem 1

By Definition 7, we know that for any ARC C_1 in A, $(x, y) \in E_{C_1} \implies (g(x), g(y)) \in E_{Agg(g)(C_1)}$.
By Definition 4, we know that $couldGetAccess(C, x, y) \iff \exists C_1 : C \vdash^* C_1 \wedge (x, y) \in E_{C_1}$.
Thus $couldGetAccess(C, x, y) \implies \exists C_1 : C \vdash^* C_1 \wedge (g(x), g(y)) \in E_{Agg_c(g)(C_1)}$.
By Lemma 2, $C \vdash^* C_1 \implies D \vdash^* Agg_c(g)(C_1)$.
Finally, by Definition 4,
$couldGetAccess(C, x, y) \implies couldGetAccess(D, g(x), g(y))$

COROLLARY 1. *Existence of tractable aggregation*
For any ARC C, there exists computable predicates $couldGetAccessAgg_C(x, y)$ and $couldGetKnowledge_C(k(s_1, \dots, s_{\mathbf{ar}(k)}))$ such that
$\neg couldGetAccessAgg_C(x, y) \implies \neg couldGetAccess(C, x, y)$ *and*
$\neg couldGetKnowledge_C(k(s_1, \dots, s_{\mathbf{ar}(k)})) \implies \neg couldGetKnowledge(C, k(s_1, \dots, s_{\mathbf{ar}(k)}))$.

Proof of Corollary 1

For any ARS $A' = (\mathbf{S}', \mathbf{P}', \mathbf{I}')$, such that $\mathbf{S}'$ is finite, $\mathbf{P}'$ and $\mathbf{I}'$ are finite. Any ARC D in A' will also be finite. Because the rules are

monotonic and confluent, we can find E, such that $D \vdash^* E$ and $\forall F : E \vdash^* F \implies E = F$ by a finite number of reductions. By the Theorem 1, we can conclude that any surjective application from $\mathbf{S}$ to a finite set, define an aggregation which is computable in finite time.

A very simple approach is to take a subset S of $\mathbf{S}$ and to define

$$\begin{array}{llll} g: & \mathbf{S} & \rightarrow S \cup \{out\} \\ & s \in P(S) & \mapsto s' \in \tilde{P}(s) \\ & s \notin P(S) & \mapsto out \end{array}$$

where $P(S) = \{s | \exists s_0, \ldots, s_n : s_0 \in S \wedge \forall 1 \leq i \leq n : (s_{i-1}, s_i) \in \mathbf{P}\}$ and $\tilde{P}(s) = \{s_0 | \exists s_1, \ldots, s_n : s_n = s \wedge \forall 1 \leq i \leq n : (s_{i-1}, s_i) \in \mathbf{P}\}$.

4. SCOLL : Safe Collaboration Language

The SCOLL language is a subset of the LP calculus extended with search. It is tractable and therefor not Turing complete. It can be made Turing-complete by making its domain countably infinite and allowing an countably infinite number of clauses.

We will define the language without using LP calculus and will show the correspondence to LP calculus only afterwards. That allows us to take advantage of the restricted definitions in SCOLL, which will improve clarity.

4.1 Definitions

A *store* over a set X named by a set P with arities ar is a set of tuples on X labeled by elements of P. The arity of a tuple in the store is determined by an arity function ar.

For example:
$\sigma = \{p_1(x_1, x_3, x_4), p_2(x_1, x_2), p_2(x_1 x_3)\}$ is a store over $\{x_1, x_2, x_3, x_4\}$ named by $\{p_1, p_2\}$
of arity ar, with $ar(p_1) = 3$ and $ar(p_2) = 2$.

We will denote $\Sigma(X, P, ar)$ as the set of all stores over X named by P with arity ar.

The *transformation* defined by $f : X \rightarrow Y$ is the function:

$$\begin{array}{lll} T_f: & \Sigma(X, P, ar) & \rightarrow \Sigma(Y, P, ar) \\ & \sigma & \mapsto \{p(f(x_1), \ldots, f(x_n)) | p(x_1, \ldots, x_n) \in \sigma\} \end{array}$$

4.2 Program structure

Let $CP = \{access, child, active\}$ be the set of configuration predicate names.
Let $KP = CP \cup \{iEmited, iCollected, rEmited, rCollected, cEndowed\}$
be the set of knowledge predicate names.
Let $BP = \{create, iEmit, iCollect, rEmit, rCollect, pEndow\}$
be the set of behavior predicate names.
Let S be the countable set of SCOLL subjects.
Let V be the countable set of SCOLL variables.
Let SP be the countable set of subject predicate names.
Let ar be the local arity application for the predicates.
Let $GP = KP \cup BP \uplus SP$ be the set of the global predicate names.

$$\begin{array}{lll} ar: & GP & \rightarrow \mathbb{N} \\ & access & \mapsto 1 \\ & child & \mapsto 1 \\ & active & \mapsto 0 \\ & iEmited & \mapsto 2 \\ & iCollected & \mapsto 2 \\ & rEmited & \mapsto 1 \\ & rCollected & \mapsto 1 \\ & cEndowed & \mapsto 1 \\ & create & \mapsto 1 \\ & iEmit & \mapsto 2 \\ & iCollect & \mapsto 1 \\ & rEmit & \mapsto 1 \\ & rCollect & \mapsto 0 \\ & pEndow & \mapsto 2 \\ & p \in SP & \mapsto n \in \mathbb{N} \end{array}$$

Let Gar be the global arity application for the predicates.

$$\begin{array}{lll} Gar: & GP & \rightarrow \mathbb{N} \\ & access & \mapsto 2 \\ & child & \mapsto 2 \\ & active & \mapsto 1 \\ & iEmited & \mapsto 3 \\ & iCollected & \mapsto 3 \\ & rEmited & \mapsto 3 \\ & rCollected & \mapsto 3 \\ & cEndowed & \mapsto 3 \\ & create & \mapsto 2 \\ & iEmit & \mapsto 3 \\ & iCollect & \mapsto 2 \\ & rEmit & \mapsto 3 \\ & rCollect & \mapsto 2 \\ & pEndow & \mapsto 3 \\ & p \in SP & \mapsto ar(p) + 1 \end{array}$$

Let $GV = V \uplus \{\alpha, \beta\}$ be the set of global variables.

A configuration is an element of $\Sigma(S, CP, Gar)$
A condition is an element of $\Sigma(V, KP \uplus SP, ar)$
A consequence is an element of $\Sigma(V, BP \uplus SP, ar)$
A rule is a pair made of a condition and a consequence.
An intent is a set of rules.
The set of all possible intents is $Intents$
An explicit behavior on $S' \subset S$ is a function $eb : S' \rightarrow \Sigma(S, BP, ar)$. The set of all explicit behaviors on S' is noted $Exp(S')$.

A SCOLL program is a 6 element tuple (b, ss, ic, si, l, s) where:
$b : S \rightarrow Intents$ is the behavior (intention) function,
$ss \subset S$ is the set of search subjects,
ic is a configuration (the initial configuration),
$si : S \rightarrow \Sigma(S, SP, ar)$ is the subjects initialization function,
$l \in \Sigma(S, CP, Gar)$ is the liveness property and
$s \in \Sigma(S, CP, Gar)$ is the safety property.

A SCOLL state is an element of $\Sigma(S, GP, Gar)$. A SCOLL result is a special SCOLL state. Characterization of SCOLL results is done in the semantics of the language. For a SCOLL program prg we will call the outcome of the program, $res(prg)$, the set of all maximal (for inclusion) SCOLL results of this program.

A SCOLL program describes the behavior and initialization from the subject point of view. To ease the definition of the semantics of a SCOLL program, we define the globalization of a SCOLL program, that is, the program seen from a global point of view.

$G(b, ss, ic, si, l, s) = (G_b(b), ss, ic, G_{si}(si), l, s)$

$$\begin{array}{llll} G_b: & S \rightarrow Intents & \rightarrow 2^{(\Sigma(S\cup GV,GP,Gar)\times\Sigma(S\cup GV,GP,Gar))} \\ & b & \mapsto \{(G_{st}(s,c), G_{st}(s,d))| \\ & & \quad \exists s \in S : (c,d) \in b(s)\} \end{array}$$

$$\begin{array}{llll} G_{st}: & S \times S \rightarrow \Sigma(V, GP, ar) & \rightarrow \Sigma(S \cup GV, GP, Gar) \\ & s, \sigma \cup \sigma' & \mapsto G_{st}(s,\sigma) \cup G_{st}(s,\sigma') \\ & \{rEmited(x)\} & \mapsto \{rEmited(s,\alpha,x)\} \\ & \{rCollected(x)\} & \mapsto \{rCollected(s,\alpha,x)\} \\ & \{cEndowed(x)\} & \mapsto \{cEndowed(s,\beta,x)\} \\ & \{rEmit(x)\} & \mapsto \{rEmit(s,\alpha,x)\} \\ & \{rCollect()\} & \mapsto \{rCollect(s,\alpha)\} \\ & \{p(x_1,\ldots,x_n)\} & \mapsto \{p(s,x_1,\ldots,x_n)\} \end{array}$$

where
$p \in (GP) \backslash$
$\{rEmited, rCollected, cEndowed, rEmit, rCollect\}$

$$\begin{array}{llll} G_{si}: & S \rightarrow \Sigma(S, SP, ar) & \rightarrow \Sigma(S, GP, Gar) \\ & b & \mapsto \{p(s, s_1,\ldots,s_n)| \\ & & \quad \exists s \in S : p(s_1,\ldots,s_n) \in b(s)\} \end{array}$$

4.3 Denotational semantics

We can consider a first order logic with a signature consisting of :
the predicates GP with the arities in Gar and
the nullary functions S. The set of variables symbols of the logic is GV

Any SCOLL state σ can be completed to an interpretation I_σ by stating that it's universe is S,

$$I_\sigma(s) = k_s : \quad \begin{array}{ll} S^0 = \{()\} & \rightarrow S \\ () & \mapsto s \end{array}$$

(The interpretation in the model of s as a nullary function in first order logic is the constant function returning s as an element of the universe),
$I_\sigma(p) = \{(s_1,\ldots,s_n)|p(s_1,\ldots,s_n) \in \sigma\}$

Since the formulas will be closed, we won't need to define the interpretation of variables.

The denotational semantics of a program (b, ss, ic, si, l, s) are given by the following formulas:
$DS(b, ss, ic, si, l, s) \equiv$
$\bigwedge_{(c,d)\in G_b(b)} (\forall^* DS_{store}(c) \implies DS_{store}(d)) \wedge DS_{store}(ic) \wedge DS_{store}(G_{si}(si)) \wedge System$
where $\forall^* F$ denotes the universal closure of F.
$DS_{store}(\sigma) \equiv true \wedge \bigwedge_{p(a_1,\ldots,a_n)\in\sigma} p(a_1,\ldots,a_n)$
$System \equiv Grant \wedge Take \wedge Create \wedge Endow$
$Grant \equiv \forall x,y,z : active(x) \wedge active(y) \wedge active(z) \wedge access(x,y) \wedge access(x,z) \wedge iEmit(x,y,z) \wedge rCollect(y,x) \implies access(y,z) \wedge iEmited(x,y,z) \wedge rCollected(y,x,z)$
$Take \equiv \forall x,y,z : active(x) \wedge active(y) \wedge active(z) \wedge access(x,y) \wedge access(y,z) \wedge iCollect(x,y) \wedge rEmit(y,x,z) \implies access(x,z) \wedge iCollected(x,y,z) \wedge rEmited(y,x,z)$
$Create \equiv \forall x,y : active(x) \wedge child(x,y) \wedge create(x,y) \implies active(y) \wedge access(x,y) \wedge created(x,y)$
$Endow \equiv \forall x,y,z : active(x) \wedge active(y) \wedge active(z) \wedge access(x,z) \wedge created(x,y) \wedge pEndow(x,y,z) \implies access(y,z) \wedge cEndowed(y,x,z)$

A SCOLL state σ is final for program (b, ss, ic, si, l, s) if there is an $e \in Exp(ss)$ such that
$I_\sigma \models DS(b, ss, ic, si, l, s)$,
$I_\sigma \models DS_{store}(G_{st}(e))$ and
there is no SCOLL state σ' such that $\sigma' \subset \sigma$ and
$I_{\sigma'} \models DS(b, ss, ic, si, l, s)$,
$I_{\sigma'} \models DS_{store}(G_{st}(e))$.

A SCOLL state σ is a SCOLL result of program (b, ss, ic, si, l, s) if:
σ is final,
$I_\sigma \models DS_{store}(l)$,
$I_\sigma \models \neg DS_{store}(s)$.

4.4 Operational semantics

For a program (b, ss, ic, si, l, s), the initial state $\sigma_0 \in \Sigma(S, GP, Gar)$ is defined by $\sigma_0 = ic \cup G_{si}(si)$.

The operational semantics of a program are defined in two parts, the propagation part and the liveness checking part: The reduction rules are from a set of states to a set of states.

For the propagation part, they are:

- Monotonicity
  $U \cup V \vdash U' \cup V$ if $U \vdash U'$
- Intentional propagation
  $\{\sigma\} \vdash \{\sigma \cup T_f(D)\}$ if
  $\exists (C,D) \in G_b(b) : \exists f : GV \rightarrow S : T_f(C) \subset \sigma$
- System propagation
  $\{\sigma = \{active(x), active(y), active(z), access(x,y), access(x,z), iEmit(x,y,z), rCollect(y,x)\}\} \vdash$
  $\{\sigma \cup \{access(y,z), iEmited(x,y,z), iCollected(y,x,z)\}\}$
  $\{\sigma = \{active(x), active(y), active(z), access(x,y), access(y,z), iCollect(x,y), rEmit(y,x,z)\}\} \vdash$
  $\{\sigma \cup \{access(x,z), iCollected(x,y,z), rEmited(y,x,z)\}\}$
  $\{\sigma = \{active(x), child(x,y), create(x,y)\}\} \vdash$
  $\{\sigma \cup \{active(y), access(x,y), created(x,y)\}\}$
  $\{\sigma = \{active(x), active(y), active(z), access(x,z), created(x,y), pEndow(x,y,z)\}\} \vdash$
  $\{\sigma \cup \{access(y,z), cEndowed(y,x,z), rEmited(y,x,z)\}\}$
- Search
  $\{\sigma\} \vdash \{\sigma, \sigma \cup G_{st}(e)\}$ if $e \in Exp(ss)$
- Safety
  $\{\sigma\} \vdash \emptyset$ if $s \cap \sigma \neq \emptyset$

And for the liveness checking part:

- Monotonicity
  $U \cup V \vdash' U' \cup V$ if $U \vdash' U'$
- Liveness
  $\{\sigma\} \vdash' \emptyset$ if $l \not\subset \sigma$

The outcome of the program is U if there is a U', such that $\{\sigma_0\} \vdash^* U'$, U' is stable for $\vdash$, $U' \vdash' U$ and U is stable for $\vdash'$.

If only a finite number of subjects, variables and predicates are used in the program, then the monotonicity of the rules guarantees that the operational semantics can give an outcome in a finite number of reductions.

4.5 Relation to LP calculus

Except for search subjects, SCOLL is a strict subset of LP. This can be seen from the denotational semantics. Liveness is the goal, safety is negative knowledge (grounded, without using variables) and can only cause failure. Other conditions are Horn clauses restricted to positive information.

The main difference is in the search subjects. We are interested in all (variations of) solutions of a problem, defined by the extensive behavior of the search subjects. Rather than deriving the liveness property, we analyze all the knowledge that can be derived from a solution.

4.6 Relation to ARS/ARC

The $child$ relation in SCOLL represents the parenthood relation $\mathbf{P}$ of the AR-system.
The $access$ relation corresponds to $\texttt{access}(x, y)$ in K_C and to E_C for an ARC C.
The $active$ relation corresponds to S_C for an ARC C.
The behavior function b corresponds to the intent $\mathbf{I}$ of the ARS without the restrictions stated in section 3.2. The G function solves these restrictions so that $G_b(b)$ corresponds to $\mathbf{I}$. The $System$ in the denotational semantics and the "System propagation" in the operational semantics correspond to the $\vdash$ relation in AR -systems.

4.7 Implementation

The first version of SCOLL is implemented in Mozart-Oz [Moz03] and described in [SJV05]. It provides no syntactic support yet, and it is merely a proof of concept at this early stage. The initial knowledge and the intent of the subjects in a configuration are input, and constraint propagators are created from these descriptions.

The propagators monotonically add information to a common constraint store, corresponding to the propagation of knowledge, behavior and access between the subjects. Explicit creation is not handled yet: all subjects therefor represent aggregates of a parent entity with all its potential offspring.

When the store has evolved to a fix point before all behavior parameters of the search subjects are defined, one parameter is picked, to be tested with the value $true$. Only when backtracking, in case no solutions were found with this parameter set to $true$, will the parameter be tested with $false$. This search strategy guarantees that the first solution found has maximal collaboration (maximum set of parameters set to $true$). For all following solutions, extra constraints are added to avoid finding sub solutions (non-maximal collaboration) of the solutions that were previously found.

The final configurations corresponding to the solutions can be visualized with GraphViz [GN00, JM04, GV05]. For a detailed account of the implementation we refer to [SJV05].

5. Using SCOLL in Practice

In this section we provide an example program that expresses the *caretaker* capability pattern for revocable authority. Alice wants to give Bob revocable authority to Carol, and therefor creates a proxy she can control: the caretaker. Given a safe approximation of Alice's and the caretaker's behavior, we want to calculate the maximum collaborative behavior for Carol, that will prevent Bob from getting irrevocable authority to Carol (e.g. direct access). Carol will be the only search subject in our problem.

Figure 4 shows the initial access graph with solid arrows. The confinement property to be guaranteed here is: Bob should not be able to get direct access to Carol. The liveness property that should not be prevented is: Bob should be able to use Carol indirectly. We therefor add Dave to the configuration: an untrusted subject only Carol initially has access to, and we will check if Bob can get access to Dave.

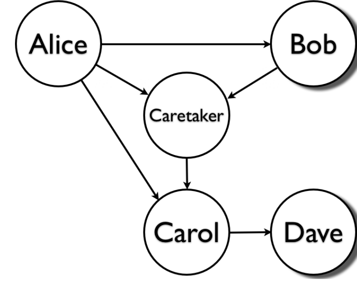

Figure 4. Initial configuration of the caretaker pattern

Since we do not know the behavior of Bob and Dave, the only safe approximation is to consider them to be completely collaborative (undefined) subjects.

This is how the the problem is programmed in SCOLL:
CaretakerPattern = (b, ss, ic, si, l, s).

$$b = \{(alice, AliceIntent), (bob, UnspecifiedIntent), (carol, SearchIntent), (ct, CaretakerIntent), (dave, UnspecifiedIntent)\}$$

The Intents of the subjects used in b are defined in Table 3

Table 3. The behavior of Alice and the caretaker

AliceIntent
$(\{\}, \{rCollect()\}),$
$\{(\{use(X)\}, \{iCollect(X)\}),$
$(\{pass(X)\}, \{rEmit(X)\}),$
$(\{use(X), pass(Y)\}, \{iEmit(X, Y)\}),$
$(\{isBob(X), isCaretkr(Y)\}, \{iEmit(X, Y)\}),$
$(\{rCollected(X)\}, \{pass(X)\}),$
$(\{isCarol(X)\}, \{use(X)\})\}$
CaretakerIntent
$(\{\}, \{rCollect()\})\}$
$(\{prxy(X)\}, \{iCollect(X)\}),$
$(\{iCollected(X)\}, \{rEmit(X)\}),$
$\{(\{prxy(X), rCollected(Y)\}, \{iEmit(X, Y)\}),$
UnspecifiedIntent
$\{(\{\}, \{iEmit(S, X), rEmit(X), iCollect(X), rCollect(), create(X), pEndow(X)\})\}$
SearchIntent
$\{\}$

$$ss = \{carol\}$$

$$ic = \{access(alice, alice), access(alice, bob), access(alice, carol), access(alice, ct), access(bob, bob), access(bob, ct), access(carol, carol), access(carol, dave), access(ct, ct), access(ct, carol), access(dave, dave), child(bob, bob), child(dave, dave), active(alice), active(bob), active(carol), active(ct), active(dave)\}$$

$$si = \{(alice, \{use(alice), pass(alice), isBob(bob), isCaretkr(ct), isCarol(carol)\}), (ct, \{prxy(carol)\})\}$$

$$l = \{access(bob, dave)\}$$

$$s = \{access(bob, carol)\}$$

5.1 The solutions

Table 4 lists the the sets of minimal restrictions corresponding to the two maximal solutions found for Carol's extensional behavior. Notice that Carol is not only prevented to return herself when being invoked. She should also never grant herself to Bob or Dave. Moreover, it is safe to either grant Alice to Bob (Dave) or grant herself to Alice, but she should not do both. The reason is that Alice, while not granting Carol initially, does not check if she gets Carol from a collaboration. Alice might therefor start "passing" Carol (to Bob) after Carol has granted herself to Alice.

Table 4. Solutions

1	Carol should not $iEmit$ herself to Alice, Bob, or Dave.
2	Carol should not $iEmit$ herself to Bob or Dave, Carol should not $iEmit$ Alice to Bob or Dave.

These results are calculated with an incomplete implementation of the language. We expect that the complete implementation – because of added expressive power – will be able to find more precise refinements of these solutions, each of them being somewhat less restricting.

Figure 5 represents the final configuration for the first solution. Dashed arrows represent access that can be reached via propagation of access, in this solution. It is clear from the graph that the confinement property and the liveness property are respected: Bob has got access to Dave but not to Carol.

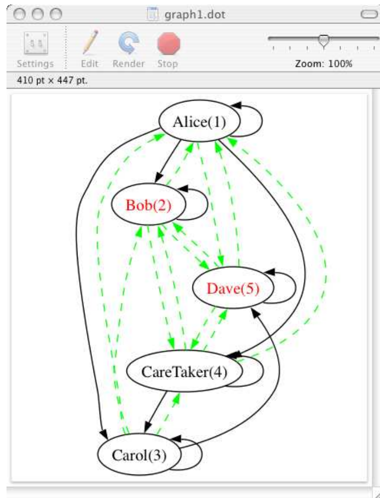

Figure 5. The graph generated from solution 1 by GraphViz

6. Conclusion and Future Work

In this paper we presented a new formalism called Authority Reduction systems to model the propagation of authority and data in capability based configurations. We showed how the classical ways to model capabilities had insufficient expressive power to be practically useful when analyzing patterns of collaborating subjects. We proved that this formalism can be used to efficiently calculate safe but precise approximations of the confinement properties in configurations where the exact calculation of confinement properties would be practically intractable. We provided a domain specific declarative language SCOLL to express the collaborative intent of subjects, the initial conditions of a configuration, and the requirements about confinement and liveness that are to be ensured. We provided the abstract syntax, operational and denotational semantics, and an initial implementation for this language, and we explained how patterns of safe collaboration can be investigated using the language.

While the caretaker pattern presented in Section 5 is interesting for its need of expressive power, it is of course in itself not sufficient to justify a complete formalism and a language. Other patterns have been used in the practice of secure programming, and they should also be analyzed to understand in what situations they can safely be applied. Examples are patterns that rely on a trusted third party to guarantee secure and non-repudiative transfer (e.g. money) can be found in [MMF00]. The sealer/unsealer pair is simple pattern that seals off authority (makes it unreachable without preventing the propagation of the sealed authority) so that only holders of the specified unsealer capability can use the sealed authority. Sealer/unsealer pairs are a crucial component in other patterns, to the degree that some capability secure languages [MSC$^+$01, Ree96] provide them as primitives in the language. Auditors [YM00] are another important mechanism that could be modeled as patterns and studied in combination with other patterns.

As mentioned in Section 3.2, and contrary to earlier, more static versions of Authority Reduction Systems [SV05b], we have not yet explicitly modeled the propagation of *data* for the formalism presented in this paper. In capability systems, preventing data to flow is a harder than preventing capabilities from being propagated, even if we only consider overt communication channels.

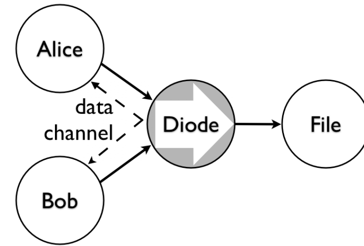

Figure 6. Overt data channel with capability diodes

Figure 6 shows an example of what we mean. Two clients Alice and Bob use a common diode D_1 that collects from its clients and emits to a file. Since AR-systems only model the *possibility* of collaborative behavior, the diode's actual acceptance of information from its clients may be influenced by these clients. Then, when this influence is observable for clients, an effective data communication channel between clients is created that is not detected by the current system. Future work will therefor explicitly model data propagation, and take these channels into account.

We plan to enhance the way confinement properties can be expressed. While we can only put restrictions on the *effects* of propagation of authority and data, it would be interesting to also be able to express requirements about the way authority and data *flows*. We are therefor investigating the use flow graphs that are *derived* from the configuration and in which these restriction can be expressed as graph reachability constraints [QVD05].

The implementation of the current version of our pattern analysis tool as described in [SJV05] is in a prototype phase and needs improvement on functionality and efficiency.

7. Related Work

The way we analyze the flow of authority in capability based patterns of collaboration bears similarity to recent work in statical [NNH99] and partially statical [Mye99] flow analysis. An impor-

tant difference and complementarity is in the scale of the problems that are analyzed, and consequently in the goal of the analysis.

For patterns of secure collaboration to be useful, we have to find and understand the boundaries of their applicability in different contexts. A pattern will typically consist of only a few roles (often less than 10), but if the language to describe the roles is very expressive, the possible ways for them to interact can be huge. From these possible ways of interacting, some have to be avoided because of their effects. The approach presented in this paper allows flexibility and arbitrary precision in the specification of the collaborative behavior of the roles in the pattern.

The results of pattern analysis are a *complete* list of possible concrete minimal sets of limitations on the behavior of the search roles in the pattern, that will each be sufficient (in an environment that complies with the capability rules) to guarantee the confinement properties for which the pattern is designed. From this list, the programmer can choose the set of limitations best suited to his needs.

The work of Guttman et al. [Jos05] models dialogs between two parties that accumulate monotonically growing knowledge, to analyze security protocols. In their setting, the accumulated knowledge can result in less cooperation as well as in more cooperation. Whereas a "simplistic" way to model protocols would have involved temporal logic to constrain the order between the events, they were able to avoid this. 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].

The insight inspired us to a solution that completely abstracts from time. We will model an extra subject for every decrease in a subject's knowledge or behavior that is to be modeled. The original subject will exhibit its behavior before the behavior decrease whereas the new one will be created when the specified conditions apply, and exhibit only the post-decrease behavior. To model the uncertainty about the actual time of the revocation, the creation of the second one will not disable the first one. This allows us to require different behavior before and after a certain "event" (e.g. the actual revocation of authority in the caretaker pattern), without modeling the event in time.

Except for our previous research in this field [SV05b, SMRS04], we have not found any recent work that focusses on formal safety analysis in capability systems.

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 Stefano Gualandi and Luis Quesada for fruitful discussions on the topics related to constraint programming in Mozart-Oz. We thank Mark Miller for his much appreciated assistance on issues of capability-based security.

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`.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[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.

[GV05] GraphViz - Graph Visualization Software, 2005. `http://www.graphviz.org/`.

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

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

[JM04] Michael Jünger and Petra Mutzel. *Graph Drawing Software*. Mathematics and Visualization. Springer, Dec 2004.

[Jos05] Joshua D. Guttman and Jonathan C. Herzog and John D. Ramsdell and Brian T. Sniffen. Programming cryptographic protocols. Technical report, The MITRE Corporation, 2005. Available 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.

[MMF00] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Financial Cryptography 2000*, Anguilla, British West Indies, February 2000.

[Moz03] Mozart Consortium. The Mozart Programming System, version 1.3.0, 2003. Available at `http://www.mozart-oz.org/`.

[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.

[Mye99] Andrew C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[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.

[Ree96] Jonathan A. Rees. A security kernel based on the lambda-calculus. Technical report, MIT, 1996.

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

[SDN+04] Jonathan Shapiro, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark 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`.

[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.

[SJV05] Fred Spiessens, Yves Jaradin, and Peter Van Roy. Using constraints to analyze and generate safe capability patterns, 2005. Submitted to the first International Workshop on Applications of Constraint Satisfaction and Programming to Security (CPSec'05). Available at `http://www.info.ucl.ac.be/people/fsp/cpsec/cpsec05.pdf`.

[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). Available at `http://www.info.ucl.ac.be/people/fsp/tgc/tgc05fs.pdf`.
Presentation available at `http://www.info.ucl.ac.be/people/fsp/auredsysfinal.mov`.

[SW00] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 166–176, 2000.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *16th Symposium on Operating System Principles*, October 1997.

[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.