Using Constraints To Analyze And Generate Safe Capability Patterns

Fred Spiessens, Yves Jaradin, and Peter Van Roy

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

Abstract. We present a dual purpose CCP application for capability based security. In a first setting, the application analyzes capability patterns of collaboration by calculating upper bounds on the propagation of (overt) causal influence. In this setting, all the relied upon restrictions in the behavior of the subjects in the pattern are input and transformed into constraint propagators.
In a second setting, the application calculates how the behavior of a (set of) trusted subject(s) in the pattern should be restricted, given the global safety properties that have to be respected.
From earlier theoretical results [SV05], we are confident that our approach is complete (all safety breaches are found and the proposed restrictions on behavior are sufficient). Because the tool is currently in a very early stage of its implementation, we only present a small set of preliminary quantitative results.

1 Introduction

In 1976 Harrisson, Ruzzo, and Ullman [HRU76] showed that the calculation of safety properties in general is an intractable problem. As their modeling language was Turing-complete, this intractability was the inevitable price to pay for its expressive power. A few years later, Take-Grant systems [BS79] were proposed for the analysis of capability based security problems [DH65]. In this model and its extensions, the safety properties are tractable [LS77,FB96], but the formalism lacks the power to express *carefully restricted* collaborative behavior. The need for a more expressive model that takes such restrictions into account is explained in [MS03,SV05].

The formal models presented in [SV05] provide the necessary expressive power to precisely model restrictions in subject behavior relevant to the propagation of information and authority between (sets of) collaborating entities. For finite configurations the calculation of the safety properties is tractable. The propagation induced by newly created entities is safely approximated by accumulating their behavior into the creating entity (parent). This result allows us to safely model unknown (untrusted) entities without considering their possible offspring. When modeling an entity's behavior, only the behavior restrictions it shares with all its potential children will be modeled as actual restrictions.

In the tool we present here, subject creation is restricted to this *implicit* form. The development is currently in a prototype stage and has lots of other limitations that will gradually be removed as the tool matures. Only the propagation of subjects is currently supported, and data propagation can only be modeled by substituting the data with non-collaborative subjects.

Section 1.1 introduces the most important capability security concepts. Section 2 gives a birds-eye overview of the tool. Design and implementation details are discussed in Section 3. Section 4 shows an example of how the tool can be used. The most important future extensions are listed in Section 5.

1.1 Glossary

Before explaining the goal and the approach of the tool, let us clarify the most important terms that will be used in this paper:

Entity : A loaded instance of a programmed entity like a procedure, an object, a process, a component or an agent. An entity can only be accessed (used) via unforgeable references (*capabilities*) that combine the *designation* of the entity with the *authority* to use the entity.

Subject : The modeled representation of an entity (possibly representing also the set of entities created at runtime by the entity, as explained earlier). Subjects could for instance be modeled from static analysis. Alternatively, subjects can be *specifications* for entities yet to be programmed (e.g. model based programming).

Subject behavior : The *willingness* of a subject to collaborate with another subject in a certain way. A subject's behavior should be a safe (over-) approximation of the behavior of the entity it models. If only the slightest possibility of collaboration exists that can lead to the entity propagating information or authority, the corresponding subject should have this behavior. There is more on subject behavior in section 2.2.

Potential Authority : The possible effects on propagation of authority and information that *could* be exerted by an entity if the entity *would be programmed* to do so. A subject's potential authority is a safe (over-)approximation of the potential authority of the entity it models.

Actual Authority : The possible effects on propagation of authority and information that can be exerted by an entity, when we take into account what is known about *its actual behavior*. A subject's actual authority is a safe (over-)approximation of the actual authority of the entity it models.

Capability rules of propagation: In pure capability systems, propagation of authority and information is only possible via either:

1. **collaboration :** an entity (the invoker, indicated by the prefix `i`) can initiate collaboration with another entity it has access to (the responder, indicated by the prefix `r`). In such a collaboration, either of them (the emitter) can provide data or subjects it has access to, to the other (the collector) if the latter is willing to collaborate in that way. It is always the emitter who decides what data or authority will be propagated, and it is always the invoker who decides what entity to collaborate with.

2. **parenthood :** New entities can only be created by entities. An entity that creates a new entity thereby gets the sole access to it.
3. **endowment :** The parent entity, upon creation of its child, endows the child with a subset of its own access.

Capability systems and their rules for (overt) propagation of influence are described in [MS03,SV05]).

Configuration : An access graph of subjects. A configuration can evolve via collaboration between its subjects. Such collaboration is governed by the capability rules of propagation and by the behavior of the subjects.

Capability Pattern : A useful configuration together with its well understood and described safety properties (access that is prevented) and liveness properties (access that is *not* prevented).

2 Overview of the Tool

2.1 The Goals

Figure 1 depicts the Caretaker pattern that will be a running example throughout this paper. *Caretaker*, created and controlled by *Alice*, is a proxy for *Carol*, to be used by *Bob*. *Alice* can order *Caretaker* to stop collaborating, in an attempt to revoke the *authority–to–invoke–Carol* she has granted to *Bob*. The fact that *Bob* and *Dave* are undefined subjects (modeling unknown entities and their offspring) is indicated by a shadow.

For the pattern to really allow revocation, *Bob* should never get direct access to *Carol* (indicated with the dashed arrow ending in a cross). Because the pattern is to be useful in a non-trivial context, we want to also make sure that *Bob* will not be prevented from getting access to *Dave* (indicated by the dashed arrow from *Bob* tot *Dave*).

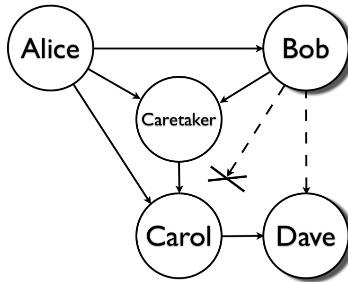

Fig. 1. The Caretaker Pattern for Revocation

In this paper, we use the term *safety property* to mean access that is effectively prevented, while *liveness property* refers to access that is *not* prevented by restrictions in the behavior of the subjects. The tool can be used for two complementary goals:

1. **Check Requirements :** From the specifications for the behavior of the trusted collaborating entities, the maximal propagation of access is calculated. The result will show directly whether or not the required safety and liveness properties are satisfied. In the example, the behavior of *Alice*, *Caretaker* and *Carol* (the trusted subjects) will be decisive.
2. **Calculate Behavior Restrictions :** Given a set of required safety- and liveness properties, what minimal sets of behavior restrictions for a certain subject can ensure the global properties? The subject(s) of which the behavior restrictions are calculated will be called query-subject(s). For multiple query subjects the minimally restrictive combinations of necessary restrictions in the behavior of the query subjects will be calculated.

In the example, the minimal sets of behavior restrictions for *Carol* can be calculated if the behaviors of *Alice* and *Caretaker* are fixed. Alternatively one could for instance calculate all possible combinations of *Alice's* and *Carol's* restrictions, given the proxy-behavior of *Caretaker*. The current version of the tool is not ready to calculate combinations of three query subjects in a reasonable amount of time. It is possible to give a minimum behavior to a query subject though, which drastically speeds up the calculation of the remaining restrictions.

2.2 Monotonic and Confluent Approach

All access to subjects (and to data) that is present in an initial configuration will be stored as simple boolean constraints in a constraint store. Every subject's initial knowledge about the configuration (its relations towards subjects and data it has access to) will also be presented as boolean constraints.

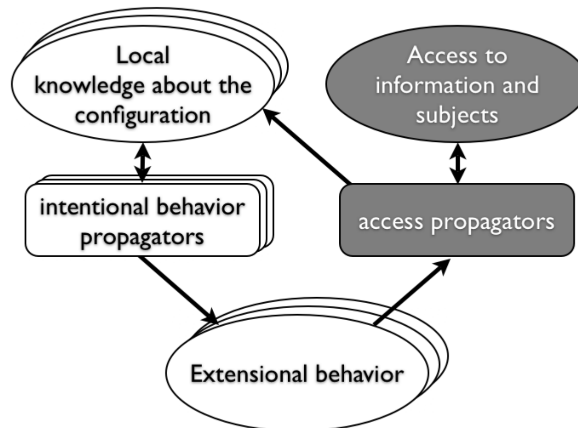

Fig. 2. The propagation of access, knowledge and behavior

A set of subject-specific propagators corresponding to the subject's intentional behavior will test (*ask*) the subject's current knowledge and generate new behavior constraints in the constraint store (boolean constraints). We call these constraints the subject's extensional behavior. Another set of propagators will model how access can propagate via collaboration in the configuration – generating new access constraints – and will also make sure that the collaborating subjects are informed of the observable effects of the collaboration, generating extra knowledge constraints.

The working of these propagators, from the point of view of a single subject, are graphically depicted in Figure 2. The system-wide propagators and constraints are gray. The constraint store will eventually reach a fix point, representing the maximum possible propagation of access in the configuration, from which goal **1.** can be inferred. If goal **2.** is pursued, a distribute-and-search process will search for (one or all) satisfactory solution(s) to the query-subject's extensional behavior.

The access constraints have the form $access(S_1, X)$ meaning that subject S_1 has access to X (X being another subject or data). The extensional behavior constraints of a subject S_1 have the following form (S_1 is an *implicit* first argument that will be made *explicit* when used by the access propagators) :

predicate	**meaning**
$iEmit(S_2, X)$	Subject S_1 is willing to invoke subject S_2, and pass X as a parameter to it.
$iCollect(S_2)$	Subject S_1 is willing to invoke subject S_2, and accept whatever S_2 provides as a return value
$rEmit(X)$	Subject S_1 is willing to return subject X upon being invoked.
$rCollect()$	Subject S_1 is willing to accept whatever input argument S_1 is being invoked with.
$rExchange(X, Y)$	Subject S_1 is willing to accept whatever input argument S_1 is being invoked with, and if it is X, then it is willing to return Y.

Notice that $rExchange()$ allows S_1 to differentiate its behavior towards its invokers, based on a proof-of-access that these invokers can provide to S_1.

The knowledge constraints generated by the access propagators toward subject S_1 have the following form (S_1 is now an *explicit* first argument that will be made *implicit* when used by subject S_1's behavior propagators) :

predicate	**meaning**
$iEmitted(S_1, S_2, X)$	Subject S_1 has successfully invoked S_1, and passed X as a parameter to it.
$iCollected(S_1, S_2, Y)$	Subject S_1 has successfully invoked subject S_2, and accepted Y as a return value from the invocation. This means S_1 has now got access to Y
$rEmitted(S_1, X)$	Subject S_1 has successfully returned subject X upon being invoked.
$rCollected(S_1, Y)$	Subject S_1 has accepted input argument Y upon being invoked. S_1 has now got access to Y.
$rExchanged(S_1, X, Y)$	Subject S_1 has returned Y on the basis of having received X *in the same invocation.*
$access(S_1, X)$	Subject S_1 has access to X, either acquired by collecting or from initial conditions.

The other knowledge constraints are subject-specific, and only the subject's intentional behavior propagators will be able to read/write to them. They will have the subject itself as an *implicit* first argument.

2.3 Input

The tool takes an initial configuration as input, consisting of an access graph of named subjects of which the intentional behavior is described. The intentional behavior of every subject is given as a set of logical implications (Horn clauses). The condition (body) of such an implication will contain knowledge-predicates, the conclusion (head) can contain subject-specific knowledge predicates and extensional behavior predicates.

For example, the proxy behavior that will characterize *Caretaker* in the Caretaker pattern could be specified using a subject-specific knowledge predicate $isMyProxy()$ in the following rules:

$$\begin{array}{ll} iEmit(S, X) & \text{:- } isMyProxy(S) \wedge rCollected(X) \\ iCollect(S) & \text{:- } isMyProxy(S) \\ rEmit(X) & \text{:- } iCollected(S, X) \\ rCollect() & \end{array}$$

Subjects are further initialized with a set of facts, that represent their initial partial knowledge (predicates) of the configuration. A subject's initial knowledge predicates will typically represent part of its relations towards the subjects (or data) it initially has access to. The access graph is also described as a set of (access) facts.

A set of safety properties and liveness properties are added to the configuration in the form of logical combinations of basic constraints (typically access constraints). The safety properties will be negated before being converted into a propagator that will cause failure upon possible violation of the property. Before a solution is validated, the liveness properties will also be verified.

If the goal is to calculate extensional behavior, the list of query subjects should be provided too.

2.4 Output

The tool calculates from the initial configuration, the maximal configuration containing all possible access. When a failure is detected, it is straightforward to construct witness traces (evidence) of how the safety properties can be violated, from the constraints that were added to the store. Upon success, the store shows the maximal extent to which data and capabilities (subjects) can be propagated. It is then simple to check the liveness requirements.

If the extensional behavior of a query subject is calculated (via search), the tool extracts for every solution, from the quiescent store corresponding to that solution, an overview of its extensional behavior predicates that are true (effectively leading to allowed collaboration), false (collaboration could lead to a violated safety property), or undefined (not relevant in the configuration).

3 CCP Based Implementation

In this section we describe how the constraint propagators are designed. Apart from the propagators for intentional subject behavior and access propagation, we present some additional propagators that will assist the calculation. We also describe our strategy for search and distribution.

3.1 Constraint Propagators

Propagators for Access These propagators are independent of the actual configuration and the specified behavior of the subjects. They are a direct representation of the way how, in capability systems, information and access are propagated via collaboration.

1. Granting: the invoker emits, the responder collects.

$$\frac{access(S_1,S_2) \wedge access(S_1,X) \wedge iEmit(S_1,S_2,X), \wedge rCollect(S_2)}{access(S_2,X) \wedge iEmitted(S_1,S_2,X) \wedge rCollected(S_2,X)} \tag{1}$$

2. Take rule: the invoker collects, the responder emits

$$\frac{access(S_1,S_2) \wedge access(S_2,X) \wedge iCollect(S_1,S_2), \wedge rEmit(S_2,X)}{access(S_1,X) \wedge iCollected(S_1,S_2,X) \wedge rEmitted(S_2,X)} \tag{2}$$

3. Exchange rule: both invoker and responder emit and collect. The responder bases his decision to emit on (obtainable knowledge about) what he collected during the invocation.

$$\frac{\begin{array}{c}access(S_1,S_2) \wedge access(S_1,X) \wedge access(S_2,Y) \wedge iEmit(S_1,S_2,X)\\ \wedge rCollect(S_2) \wedge iCollect(S_1,S_2) \wedge rExchange(S_2,X,Y)\end{array}}{\begin{array}{c}access(S_2,X) \wedge access(S_1,Y) \wedge iEmitted(S_1,S_2,X) \wedge rCollected(S_2,X)\\ \wedge iCollected(S_1,S_2,Y) \wedge rExchanged(S_2,X,Y)\end{array}} \tag{3}$$

Using the propagators (1) and (2) we can reduce (3) to:

$$\frac{iEmitted(S_1, S_2, X) \wedge access(S_2, Y) \wedge iCollect(S_1, S_2) \wedge rExchange(S_2, X, Y)}{access(S_1, Y) \wedge iCollected(S_1, S_2, Y) \wedge rExchanged(S_2, X, Y)} \quad (4)$$

Behavior Propagators These represent the subject-specific reaction to positive knowledge about access to subjects and data, and about the way this access was acquired. They can refine knowledge and use knowledge to generate behavior. They are restricted in the sense that they cannot generate new access (that would defy the capability rules) or knowledge of the kind that is produced by the access propagators. To reflect the fact that subjects can only refine their own knowledge, and generate their own behavior, these propagators will also be restricted in scope. The first argument of every predicate is *implicit* and designates the subject who's behavior is being described. It will become *explicit* only for the access-propagators and for the assisting propagators.

A Horn clause that partially describes subject S_1's behavior like this:

$$behavior(B_1, \ldots B_n) \leftarrow condition_1(C_{1,1} \ldots C_{1,k}) \wedge \ldots \wedge condition_j(C_{j,1}, \ldots C_{j,m}) \quad (5)$$

... will be converted into a propagator like this:

$$\frac{condition_1(S_1, C_{1,1}, \ldots C_{1,k}) \wedge \ldots \wedge condition_j(S_1, C_{j,1}, \ldots C_{j,i})}{behavior(S_1, B_1, \ldots B_n)} \quad (6)$$

The behavior for an unknown (untrusted) subject S_1 can be represented with a single propagator:

$$\frac{true}{iCollect(S_1, S_2) \wedge iEmit(S_1, S_2, X) \wedge rCollect(S_1) \wedge rEmit(S_1, X)} \quad (7)$$

Assisting Propagators As soon as one of two "unknown" (completely collaborative) subjects has direct access to the other one, they will inevitably end up sharing the same access. Therefore, the query subject should not even consider treating these subjects differently, as it will not have a different effect. For every pair of unknown subjects, (S_1, S_2), and for every query subject S_q, we will add the following propagators:

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{iEmit(S_q, S_1, X) = iEmit(S_q, S_2, X)} \quad (8)$$

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{iEmit(S_q, S, S_1) = iEmit(S_q, S, S_2)} \quad (9)$$

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{iCollect(S_q, S_1) = iCollect(S_q, S_2)} \quad (10)$$

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{rEmit(S_q, S_1) = rEmit(S_q, S_2)} \tag{11}$$

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{rExchange(S_q, S_1, X) = rExchange(S_q, S_2, X)} \tag{12}$$

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{rExchange(S_q, X, S_1) = rExchange(S_q, X, S_2)} \tag{13}$$

Without going into details about the implementation, it is easy to see how these propagators can be efficiently implemented: as far as the query subjects are concerned, their extensional behavior can consider both subjects as one aggregate subject. This principle can also be used in a weaker form for any two subjects, when it would be useless for the query subjects to differentiate (a particular part of) their behavior towards the one or the other. It is a form of symmetry braking in the constraint model.

Safety properties are mere boolean constraints that are set to *false*, to cause failure when they are unified with *true* by a propagator. We are experimenting with propagators for safety properties in the form (14) and (15), to promote early failure detection. Propagating *false* to a query subject's extensional behavior constraints will decrease the depth of the search tree.

$$\frac{\neg access(S_q, X)}{\neg(access(S_1, S_q) \wedge access(S_1, X) \wedge iEmit(S_1, S_2, X) \wedge rCollect(S_2)} \tag{14}$$

$$\frac{\neg access(S_q, X)}{\neg(access(S_q, S_1) \wedge access(S_1, X) \wedge rEmit(S_1, X) \wedge iCollect(S_1, S_2)} \tag{15}$$

3.2 Constraint Implementations

We implement the tool in the Mozart environment [Moz03] for the multi-paradigm language Oz [Smo95,VH04], which provides strong support for concurrent constraint programming [Sch02]. Because the implementation of the basic boolean constraints can have a big impact on the efficiency of the propagators mentioned above, we are currently experimenting with two approaches in parallel, one using finite domain integers and the other one using finite sets of integers. We give a short description of both.

Finite Domain Integer Constraints Every predicate is modeled as a finite domain integer variable in a domain ranging from 0 ($false$) to 1 ($true$). Logical connectives can now be implemented as a product (logical and) or as a sum (logical or, using the appropriate domain for the sum). Propagator (16) shows how we implement the safety property propagator (14) with finite domain integers and the *sum* and <: (strictly smaller) propagators.

$$\frac{\neg access(S_q, X)}{sum([access(S_1, S_q), access(S_1, X), iEmit(S_1, S_2, X), rCollect(S_2)]) <: 4} \tag{16}$$

To avoid a combinatorial explosion of the number of finite domain variables, we implement logical *and* with nested implications *impl* where appropriate. The implication propagator will wait for its condition to be true, before telling its conclusion. The conclusion can again be an implication. This is how we implement the access propagators in this approach. Propagator (17) gives an example of how the granting propagator (1) is implemented.

$$\begin{aligned}
&impl(\ access(S_1,S_2),\\
&\quad impl(\ access(S_1,X),\\
&\quad\quad impl(\ iEmit(S_1,S_2,X),\\
&\quad\quad\quad impl(\ rCollect(S_2),\\
&\quad\quad\quad\quad (access(S_2,X)\ \wedge iEmitted(S_1,S_2,X)\\
&\quad\quad\quad\quad\quad \wedge rCollected(S_2,X)\ )))))
\end{aligned} \tag{17}$$

Finite Sets Constraints In this approach we present an n-ary predicate as the finite set of all n-tuples of subjects that satisfy the predicate. We assign a unique integer to each n-tuple of subjects, to represent that tuple in the set. Implications over predicates are translated into set inclusions over the corresponding finite sets of integers, disjunction is translated to union, and conjunction becomes intersection. Of course, these operations should only be performed on compatible predicates.

In (18) and (19) we consider two clauses in *Caretaker's* behavior to explain how the predicates are made compatible.

$$iEmit(S,X) :- \quad isMyProxy(S) \wedge rCollected(X) \tag{18}$$

$$rEmit(X) :- \quad iCollected(S,X) \tag{19}$$

In the body of clause (18) we cannot simply use set intersection, because $isMyProxy(S)$ and $rCollected(X)$ have a different variable. First we have to make the cartesian product in the following way:
$iEmit(S,X)$:- $(isMyProxy(S) \times isSubj(X)) \wedge (isSubj(S) \times rCollected(X))$
We use $isSubj()$ as a unary predicate that is true for every subject. The clause now translates to the finite set propagator:
$iEmit \subseteq (isMyProxy \times isSubj) \cap (isSubj \times rCollected)$.
The cartesian product of finite sets is implemented by recalculating the individual integers (tuples) of the result set.

The head of clause (19) has one less variable than its body. Therefore we use a projection operation $P_{(\ldots)}$ that extracts a sub-tuple from every tuple in the predicate, and we convert the clause to: $rEmit(X)$:- $P_{(X)}(iCollected(S,X))$. This translates to the finite set propagator: $rEmit \subseteq P_{(2)}(iCollected)$.

All clauses can thus be translated to finite set propagators using the proper combination of cartesian product, projection, inclusion, union, and intersection. Because the cartesian product is the most costly operation, we try to minimize its use and the size of its argument sets. For instance, the actual implementation of the clause (19) will be simplified to: $iEmit \subseteq isMyProxy \times rCollected$.

Preliminary Comparison The current state of the tool does not yet allow us to make quantified comparisons or draw conclusions about which of the two approaches is best suited for what kind of problems. We provide for both approahes the preliminary benchmark results for the calculation of *Carol's* restriction in the caretaker pattern as described in Section 4. We currently believe that the optimal overall approach will be a merge of both.

The finite domain integers approach finds the 4 solutions in 5 seconds, using 318 search nodes (not failed neither succeeded nodes), with a search tree depth of 35.

The finite set approach finds the first three solutions after 1, 20, and 63 seconds (total time) respectively. The forth solution was not yet found after 30 minutes. The finite set approach finds the first tree solutions using 440, 6000, and >12400 search nodes with a maximal search tree depth of 37.

All calculations were done on a 1.25 GHz PowerPC G4 with 1 GB memory.

3.3 Search and Distribution Strategies

When testing suitable behavior for query subjects, we use a depth first search strategy. To detect failures as fast as possible, we order the extensional behavior constraints of the query subject(s) by the number of concurrent constraint propagators that are currently waiting for that basic constraint to become determined. This functionality is implemented in the `FD.reflect.nbSusps` built-in procedure. The distribution stops when no more behavior aspects have at least one propagator waiting for it, indicating that all feasible ways for propagating access have been exhausted.

To detect the maximal solutions first, we always try the 1-alternative (*true*) first (indicating willingness to collaborate).

Further symmetry breaking is done via a particular use of the branch-and-bound facilities provided by the environment (with our thanks to Raphaël Collet for pointing out this possibility). Whenever we find a solution, we add it to a list of currently found solutions, and then tell a constraint that the next solution should not be a sub-solution of any solution in the list. Sub-solutions are solutions with less-than-maximal relevant collaboration properties. The branch-and-bound constraint works like this:

```
proc{NoSubSolutions Recent Next}
   Recent.oldSols := (Recent.solution)|@(Recent.oldSols)
   {ForAll @(Recent.oldSols)
    proc{$ Traces#_}
       {FD.sum {Map {Filter Traces fun{$ Tr} Tr.value==0 end}
                fun{$ Tr} {GetPred Tr.subjId Tr.pred Next} end}
        ´>:´ 0}
    end}
end
```

The procedure `NoSubSolutions` adds a propagator that ensures that there will be at least one non-collaborative (= 0) relevant behavior predicate of every previous solution that will be collaborative (= 1) in the next solution. Together

with the choice strategy “try the 1-alternative first”, this ensures that no sub-solutions are found or searched for.

4 Example

As an example we calculate *Carol's* necessary behavior restrictions in the ”caretaker” pattern, introduced in Figure 1 of Section 2.

Since we don't know the behavior of *Bob* and *Denis*, the only safe approximation is to consider them to be completely collaborative (unknown) subjects. The intentional behavior of *Alice* and *Caretaker* is listed in table 1, together with their initial access and knowledge.

Table 1. The behavior of *Alice* and *Caretaker*

Alice	
$iEmit(S,X)$:- $use(S) \wedge pass(X)$ $iEmit(S,X)$:- $isBob(S) \wedge isCaretaker(X)$ $iCollect(S)$:- $use(S) \wedge pass(S)$ $rEmit(X)$:- $pass(X)$ $rCollect()$:- $true$	knowledge → behavior
$pass(X)$:- $rCollected(X)$ $use(X)$:- $isCarol(X)$	knowledge → knowledge
$access(1) \wedge isSelf(1) \wedge use(1) \wedge pass(1)$ $access(2) \wedge isBob(2)$ $access(3) \wedge isCaretaker(3)$ $access(4) \wedge isCarol(4)$	initial knowledge
Caretaker	
$iEmit(S,X)$:- $isMyProxy(S) \wedge rCollected(X)$ $iCollect(S)$:- $isMyProxy(S)$ $rEmit(X)$:- iCollected(S, X) $rCollect()$	knowledge → behavior
$access(3) \wedge isSelf(3)$ $access(4) \wedge isMyProxy(4)$	initial knowledge

Table 2 lists the solutions found for *Carol's* extensional behavior.

The first two solutions restrict *Carol's* behavior towards Alice, because *Alice* could inadvertently allow *Carol* to be collected from here by *Bob*. The precautions taken in *Alice's* behavior do not exclude this: the sixth clause in *Alice's* behavior shows that when she collects *Carol* upon being invoked, she will pass her on.

The last two solutions are not very interesting, since they don't allow *Carol* to accept any capabilities upon being invoked. An inspection of the store showed us that in these two cases, *Alice* (rather than *Carol*) is responsible for providing *Bob* access to *Denis* (the liveness property).

Table 2. solutions

1	Carol should not $rEmit$ herself. Carol should not $iEmit$ herself to Alice, Bob, or Denis.
2	Carol should not $rEmit$ herself or Alice Carol should not $iEmit$ herself to Bob or Denis, Carol should not $iEmit$ Alice to Bob or Denis.
3	Carol should not $rEmit$ herself Carol should not $iEmit$ herself to Denis, Carol should not $iCollect$ from Denis. Carol should not $rCollect$.
4	Carol should not $rEmit$ herself Carol should not $iEmit$ herself to Bob or Denis Carol should not $rCollect$.

The *exchange()* predicate was not used in the example, because we did not yet have a stable and reliable implementation for it.

5 Future Extensions

Expressive Power

Adding data : We will soon add support for data. The current solution uses non-cooperative subjects as data and does not allow to reason about the more unexpected and indirect ways in which information can flow. For instance, when a client can influence the behavior of a server, and that behavior is visible to another client, information can flow from one client to the other. Our theoretical model allows to reason about this kind of data flow, and so should our tool.

Exchange : $rExchange()$ is a recent addition of which we have to explore the possibilities and limitations. We would like to use it for enabling the *Caretaker* proxy to decide its collaboration per invocation, but invocation-based granularity can easily lead to a combinatorial explosion.

Derived safety properties : It is better to reason about the *flow* of authority than only about the *effects* of authority propagation. This can be done in flow-graphs (with arcs representing the direction of the flow) that are derived from the configuration. We will use reachability constraints [QVD05] in derived flow graphs to express more elaborated safety properties. We will then be able to express the more precise safety property for the caretaker pattern: "Carol's authority should be *reachable* for Bob only via the caretaker".

Functionality

Calculate intentional behavior : To provide a real specification for the query subject's intentional behavior we have to derive such specifications from the extensional behavior of the query subjects.

Real pattern generation : We want to experiment with adding trusted subjects to a configuration in critical places, when no safe solutions can be found in a pattern. This would allow us to generate patterns from high-level specifications.

Performance and Scalability

Add pruning : The propagators for the safety-properties should help pruning the search space more than they do now.

Merging the two approaches : The approach based on finite sets has not yet been optimized to the level of the finite integer domain approach. We need to take care of that, and then measure the performance for different kinds of problems to find out which of the two approaches in Section 3.2 is the most performing and scalable, and how we can merge them to get the best of both worlds.

User Interface

Write a parser : Currently we input the problems directly in parsed form.

Web interface : The tool would be useful to the community of capability developers. Therefore we want to wrap it into a web application.

Integrating GraphViz : We have an ad-hoc connection to the GraphViz tool [GN00,KN93] for visualizing the graphs generated by the solutions. We will properly integrate this visualization tool.

6 Related Work

Whereas actual applications of CCP to security are not widespread yet, we see a few interesting opportunities that are related to model checking and pattern generation as we describe it in our paper.

The work of Joshua Guttman et al. [Jos05] uses a datalog-like language for secure protocol design . We believe that that by using constraints (and search) the way we do, that approach could be enriched to also support the "generation" of such protocols, from general descriptive rules.

Jan Jürjen's work on security specifications in UML [JÖ5] – again a model-based approach – could probably also benefit from extensions with constraint-based model checking.

7 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 Raphaël Collet for discussing the formal aspects of the model. We thank Mark Miller for his advice about capability-based security. We thank the reviewers for their useful comments and suggestions.

References

[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`.

[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.

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

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

[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. Availalbe at `http://www.ccs.neu.edu/home/guttman/`.

[KN93] Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot*. Murray Hill, NJ, 1993.

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

[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.

[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.

[Sch02] Christian Schulte. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

[Smo95] Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.

[SV05] 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 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.