# SCOLL and SCOLLAR

## Safe Collaboration based on Partial Trust

Fred Spiessens, Yves Jaradin, and Peter Van Roy

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

**Abstract.** When practicing secure programming, it is important to understand the restrictive influence programmed entities have on the propagation of authority in a program. To precisely model authority propagation in patterns of interacting entities, we have generalized an earlier formalism [SV05b] into "Knowledge behaviour Models" (KBM). To describe such patterns, we present a new domain specific declarative language SCOLL (Safe Collaboration Language), which operational semantics are expressed by means of KBMs.
To interpret SCOLL patterns we have built SCOLLAR: a model checker and generator based on constraint logic programming. SCOLLAR not only indicates whether the safety requirements are guaranteed by the restricted behaviour of partially trusted subjects, but also lists the different ways in which the behaviour of a trusted entity can be restricted to guarantee the safety properties without restricting its required functionality and (re-)usability. The tool helps programmers to build correct trusted components that can safely interact with partially trusted and untrusted components.

## 1 Introduction

Since 1976 we know [HRU76] that the calculation of safety properties in general is not computable and we have to look for safe *approximations* of safety instead. But even when using relatively simple models with tractable safety properties, *safe* approximation of safety is not trivial. In a paper on Take-Grant systems [BS79], Bishop and Snyder showed that there can be "de-facto" propagation of authority between subjects, even when no actual propagation of "de-jure" permissions takes place.

Suppose we design a distributed (or concurrent) program that will integrate two non-trusted components X and Y at runtime. X needs access to a screen for its purposes, and Y needs access to the internet. We want to keep X from getting access to the internet and Y from getting access to the screen, to prevent either of them to mount a phishing attack: tricking the user into thinking she is surfing on her bank site, when she is actually connected to the phising site.

Simply preventing X from having direct access to the internet (and Y from directly accessing the screen) is not enough. When X and Y can collaborate,

there is no way to prevent such an attack: X can provide the screen services for Y, or Y can provide the internet services for X. Some trusted components in our system might need access to both X and Y, and we want to make sure *by design*, that these trusted components do not introduce X and Y to each other. As X and Y can run in different threads (or even in different processes or on different networked machines) we cannot use a protection mechanism that is based on controlling a single call-stack (stack-walking) [WBDF97] to detect what component is trying to invoke (directly or indirectly) what other component.

In this paper we present an approach for analyzing maximal authority propagation in a configuration of collaborating entities, and a tool, based on that approach, to calculate safe alternatives for the behaviour of trusted entities in that configuration. We define *authority* as the power of a subject (a model for a programmed entity like a procedure, object, or component) to use the system's resources, regardless of how this power is exerted (what series of permissions are used or delegated). We define *collaboration* as interaction between two subjects whereby both subjects have the power to restrict and prevent the interaction.

We concentrate on the role the subjects themselves play, and on the additional restrictions their behaviour can impose to the propagation of authority in the presence of an arbitrary protection mechanism. The tool uses only the protection mechanism the user describes. We believe that such analysis of authority propagation can substantially contribute to the overall protection in a system.

In systems where subjects have no permissions by default (permissions are only provided through interaction), behaviour restriction of trusted subjects can be an effective and efficient safety mechanism. Systems for mobile agents often provide this option and depend on it to confine untrusted agents. Many memory-safe programming languages can restrict the default permissions that are provided when code is loaded, and some are designed to avoid all ambient authority (authority that is available without collaboration) by default [MSC+01,SV05a].

In section 2 we give an extensive and motivated but informal account of the basic concepts of our approach. Section 3 gives a brief formal account of our model for collaboration and safe approximation of authority propagation. Section 4 describes the syntax and operational semantics of the SCOLL language we use to specify patterns of collaborating subjects with restricted behaviour. The SCOLLAR tool for analyzing the safety properties in such patterns, and for generating safe alternatives for the behaviour of trusted subjects, is presented in Section 5. We discuss related research and future work in section 6.

## 2 Basic Concepts and Assumptions

Our tool has to serve a practical purpose: help designers and developers to understand the required behaviour restrictions for (sets of) programmed entities, given a global safety goal and a pattern (context) in which these entities will play a well-defined role.

We model collaboration and propagation in a form that matches how objects communicate via messages, and how first class procedures (closures) communi-

cate by invocation : exchanging input and output arguments that can themselves be objects or procedures. Defining new closures and instantiating new objects, is modeled by subject creation.

Many systems can effectively restrict collaboration. In memory safe programming languages, references (subjects) cannot be forged and collaboration is only possible if such a reference is available. Systems protected by permission control (reference monitoring) can impose further (or other) restrictions. While such restrictions are necessary to make our tool useful, we do not model them implicitly. Instead, the user can define rules for them, as will be explained in Section 2.4.

### 2.1 Subject behaviour

We first distinguish two major roles in a collaboration between subjects : the *invoker* (invoking procedure or sending object) who initiates the collaboration, and the *responder* (invoked procedure or receiving object). The invoker decides what subject to invoke.

Orthogonal to these roles, a subject can be an *emitter* or/and a *collector* in a collaboration. The emitter models either an invoking procedure that provides an input argument for the invocation, or an invoked procedure that provides an output argument (return value). The other party in the collaboration can then collect this emitted subject. The emitter decides what subject to emit.

We allow both invoker and responder to decide for themselves which of these roles (or none, or *both*) they want to play in a collaboration. Whether a collaboration succeeds or not, will depend on the modeled system restrictions (Section 2.4). To model real collaboration, these restrictions are expected to at least enforce complementary roles for the invoker and the responder : one collects when the other emits.

To model creation of new closures (or objects) we distinguish the *parent* and *child* roles. In many systems a parent can pass on subjects to its children without the need for the child to collaborate. Consider for instance a lexically scoped language, in which newly created (inner) closures can directly be given access to (part of) the creating (outer) scope. We call this *endowment* of the child by its parent, and it is the only form of subject propagation without collaboration that is specified by subject's behaviour.

In most systems, the parent will also automatically get a reference to its created child, without the need for any propagation at all: this will be referred to as *parenthood*. The system rules (Section 2.4) should model the effects of creation and endowment according to the corresponding effects in the modeled system.

Subject behaviour is expressed with the predicates listed in Table 1. The first argument is restricted to the subject who's behaviour is expressed. The prefix `i` indicates the *invoker* role and `r` indicates the *responder* role. The prefix `p` indicates the *parent* role when new subjects are created, and the prefix `c` (not present in table 1) will indicate the *child* role.

It is important to keep in mind that we model only a *safe approximation* of the actual behaviour of the subjects. For instance, $iEmit(S_1, S_2, X)$ means: what

**Table 1.** Predicates for subject behaviour

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

we know about (and model from) $S_1$'s behaviour does *not exclude the possibility* that $S_1$ tries to invoke $S_2$ with input argument $X$.

### 2.2 Subject Knowledge

A subject can differentiate its behaviour towards its potential collaborators, based on the knowledge it has about them. Such knowledge can be built-in (initial knowledge) or it can be acquired during collaboration (knowledge from experience). For instance, a subject could decide to provide maximal collaboration to subjects it trusts, and to trust every subject it collected from the trusted subjects it invoked. Knowledge about subjects is modeled as predicates.

Table 2 lists the knowledge predicates that can become available to a subject *from successful collaboration*. The first argument references the subject who's knowledge is expressed.

**Table 2.** Predicates for subject knowledge

| predicate | comments |
|---|---|
| $iEmitted(S_1, S_2, X)$ | $S_1$ invoked $S_2$ and emitted $X$ to it |
| $iCollected(S_1, S_2, X)$ | $S_1$ invoked $S_2$ and collected $X$ from it |
| $iExchd(S_1, S_2, X, Y)$ | $S_1$ invoked $S_2$ and emitted $X$ to it, and collected $Y$ in the same invocation. |
| $rEmitted(S_1, X)$ | $S_1$ emitted $X$ to an invoker ($S_2$) |
| $rCollected(S_1, X)$ | $S_1$ collected $X$ from an invoker |
| $rExchd(S_1, X, Y)$ | $S_1$ collected $X$ from an invoker, and emitted $Y$ in the same invocation. |
| $pCreated(S_1, S_2)$ | parent $S_1$ created $S_2$ |
| $pEndowed(S_1, S_2, X)$ | parent $S_1$ endowed its child $S_2$ with $X$ |
| $cEndowed(S_1, X)$ | child $S_1$ was endowed with $X$ |

Since we only model *safe approximation* of the actual behaviour of the subjects, the knowledge arising "from experience" in or model, is not strict knowledge : it represents the fact that we *cannot exclude that the subject can actually*

*reach this knowledge*. For instance, $iEmitted(S_1, S_2, X)$ means: we cannot exclude the possibility that subject $S_1$ succeeds in invoking subject $S_2$ and thereby emitting $X$ to $S_2$.

### 2.3 Conditional behaviour and Derived Knowledge : Subject Rules

Subjects now have knowledge and behaviour, and we want to express how their behaviour depends on their knowledge. To model a safe (over) approximation of an entity's behaviour, we demand that this dependency is monotonic: more knowledge should not lead to less behaviour. *We will explain later how non-monotonic behaviour can be modeled anyway, e.g. to express revocation* The monotonic dependency is expressed for a subject $s_1$ with simple implications of the form:

$know_1(s_1, \ldots S_i) \wedge \ldots \wedge know_n(s_1, \ldots S_j) \Rightarrow behave(s_1, \ldots S_k)$

We call the implications that define conditional behaviour *subject rules*. We denote $s_1$ in lowercase, to indicate that it is a constant, representing the subject who's behaviour is being defined here. Al other arguments are (logic) variables ranging over the domain of subjects.

**Private Knowledge** Subject rules can also derive private knowledge:

$know_1(s_1, \ldots S_i) \wedge \ldots \wedge know_n(s_1, \ldots S_j) \Rightarrow knowPriv(s_1, \ldots S_k)$

Private knowledge is knowledge that is available to the subject $s_1$ only. It is denoted with unique, user defined predicates. Think of private knowledge as a combined and contracted from of knowledge. Private knowledge is also useful to model the use of local variables inside a procedure: the fact that an entity gets assigned to a local variable is modeled as a relationship (knowledge predicate) with that entity.

If it is clear from the context that we are defining behaviour for $s_1$, we will drop the first argument of every predicate in the rule. For concise notation the right part of the rules can be a conjunction of predicates.

**Example** Here is simple example of a subject's behaviour, in concise notation.

$$trust(A) \Rightarrow iEmit(A, X) \quad (1)$$
$$trust(A) \Rightarrow iCollect(A) \quad (2)$$
$$secret(X) \Rightarrow rExch(X, Y) \quad (3)$$
$$trust(A) \wedge iCollected(A, X) \Rightarrow trust(X) \quad (4)$$

When invoking, the subject maximally emits to (1) and collects from (2) trusted subjects. When responding, it emits maximally to invokers that emit a secret in the same invocation (3). The subject will trust every subject it collects by invoking a trusted subject (4).This subject has to be initialized with some $trust()$ and $secret()$ predicates, before it will be able to collaborate.

### 2.4 System Rules

*System rules* specify when behaviour succeeds and what the effects will be. Successful behaviour is what *can not with certainty be prevented from occurring.*

System rules are dual to subject rules: they derive knowledge from subject behaviour and system knowledge. System knowledge is available only to the system (and *not* to the subject in the first argument of the predicate). The predicates in system rules cannot use shortcut notation: all arguments are (logic) variables

The system rules serve a second purpose: to model the protection mechanism in the system the user wants to analyze. To show how both concerns combine, we give here a realistic example of a consistent set of system rules, divided into two parts for clarity.

**Example (part 1)** Collaboration Rules :

Let us use a system knowledge predicate: $access(A, B)$, to indicate the permission of $A$ to both invoke $B$ and emit $B$. During collaboration, this *permission to use and pass a specific subject* will be propagated from the emitter to the collector, and both parties will be informed (given knowledge) about the consequences of their behaviour.

1. $iEmit(A, B, X) \wedge access(A, B) \wedge access(A, X) \wedge rCollect(B)$
   $\Rightarrow access(B, X) \wedge iEmitted(A, B, X) \wedge rCollected(B, X)$
2. $iCollect(A, B) \wedge access(A, B) \wedge rEmit(B, X) \wedge access(B, X)$
   $\Rightarrow access(A, X) \wedge iCollected(A, B, X) \wedge rEmitted(B, X)$
3. $iEmitted(A, B, X) \wedge access(B, Y) \wedge rExch(B, X, Y) \wedge iCollect(A, B)$
   $\Rightarrow access(A, Y) \wedge iExchanged(A, B, X, Y) \wedge rExchanged(B, X, Y)$

Rule 1 is applicable when the invoker ($A$) is also the emitter. The invoker needs access to the responder ($B$) and to the emitted subject ($X$) and the responder must be willing to play the complementary collector role. As a result $B$ gets access to $X$ and $A$ and $B$ are informed about their part in the successful collaboration.

Rule 2 describes the dual mechanism of rule 1. Now the invoker ($A$) is collecting, the responder ($B$) needs access to the emitted subject $X$, and the collaboration results in new access from $A$ to $X$.

Rule 3 shows the meaning of $rExch()$ behaviour. It allows the responder ($B$) to decide wether it wants to emit $Y$, based on what it collected ($X$) from the invoker ($A$).

**Example (part 2)** Creation Rules :

We use another system predicate: $child(A, B)$, to indicate that $A$ can create $B$. Rule 1 models *parenthood*: it gives the parent the *permission to use and pass its child*. Rule 2 models *endowment*: it enables the parent to propagates access permissions without the collaboration of the child.

1. $pCreate(A, B)child(A, B) \Rightarrow access(A, B) \wedge pCreated(A, B)$
2. $pEndow(A, B, X) \wedge access(A, X) \wedge pCreated(A, B)$
   $\Rightarrow access(B, X) \wedge pEndowed(A, B, X) \wedge cEndowed(B, X)$

An interesting class of systems that are modeled exactly by the rules in this example, is described in [MS03].

### 2.5 Patterns of Collaborating Subjects

A pattern of collaborating subjects is a set of subjects, with zero or more behaviour rules for every subject (Section 2.3), together with a set of system rules (Section 2.4) and an set of initialization predicates. Initialization predicates are grounded facts that seed the subject rules (subject knowledge and private subject knowledge predicates) or the system rules (system knowledge).

### 2.6 Restrictions

Our approach has several restrictions, some of which we plan to remove in the future.

**Multiple arguments :** We model only one input and/or one output argument per invocation, and express more complex invocations by multiple simple ones. This approximation restricts the power of the tool to express fine-grained behaviour, and we will refine this model in future versions.

**Modeling data :** We currently have no separate way to represent data, and propose to model data as a passive subject (no collaborative behaviour) if necessary. This is not sufficient, because data can sometimes propagates in ways subjects cannot. Two untrusted subjects having access to a common trusted subject can communicate data between them if one can influences the trusted subject 's behaviour (by collaborating) and the other one can observe (the modulation of) this influence.

**Non monotonic changes in behaviour :** Our monotonic approach does not allow us to directly model a subject that changes its behaviour in a non-monotonic way (e.g. by using less or completely different rules) when it becomes aware of an event (knowledge). We want to keep the monotonic approach at the subject level, as it can guarantee that our models can be expressed with arbitrary many subjects and rules, approximating reality better and better, and still be computable (see Section 3).
Instead we propose to model the entity as two subjects. One models the behaviour before the change, the other one after the change, but the rules of the latter will all have the change-triggering knowledge as an extra precondition to ensure that it is completely passive before the change is triggered. The subjects having access to the composed subject do not need to be split up. By carefully adapting the system initialization (making sure they have access to both parts of the decomposed subject, and they have no initial knowledge to differentiate between the parts), their behaviour will automatically apply to both subjects, as if it was just one from their point of view.

## 3 The Knowledge behaviour Model

In this section we define a formal model for the approach we explained. It proves that our approach allows us to model arbitrary *sets of entities* as one composed subject, and still have a safe approximation for the modeled system. We call such

multi-entity subjects *aggregates*, and the principle and mechanism *aggregation*. Aggregation ensures us that we can make safe *and computable* approximations, simply by using only a finite set of subjects and predicates.

Aggregating entities with their (potential) children can be useful and avoids the need to model and calculate creation rules. In fact, every pattern that is safe for a certain safety property, and uses only subjects that do not create, will also be safe if the subjects *do* create, as long as their offspring are in no circumstances more collaborative than the original subject.

Depending on the purpose, it can be interesting to aggregate all entities of the same behaviour, the same security clearance level, or a similar position in an access graph (e.g. between two subjects that need to be kept confined).

**Definition 1 (KBM).** *Knowledge behaviour Model :*
*A KBM is a tuple* $< S, K, B, ar, Subj, Sys >$ *such that:*
$S$ *is a set of subjects.*
$K$ *is a set of knowledge predicate labels (e.g. the ones defined in table 2).*
$B$ *is a set of behaviour predicate labels (e.g. the ones defined in table 1).*
*Let* $Q = K \uplus B$
$ar : Q \to \mathbb{N}$ *be an arity function for the predicate labels.*
*Let* $\mathbf{K} = \{q(s_1, \dots s_{ar(q)}) | s_i \in S, q \in K\}$
*Let* $\mathbf{B} = \{q(s_1, \dots s_{ar(q)}) | s_i \in S, q \in B\}$
*Let* $\mathbf{P} = \mathbf{K} \uplus \mathbf{B}$ *be the set of propagation predicates over* $S$
$\forall s \in S$ : *Let* $\mathbf{K_s} = \{q(s_1, \dots s_{ar(q)}) | s_i \in S, s_1 = s, q \in K\}$
*Let* $\mathbf{B_s} = \{q(s_1, \dots s_{ar(q)}) | s_i \in S, s_1 = s, q \in B\}$
*Let* $\mathcal{F}(A, B)$ *denote the set of functions from* $A$ *to* $B$
$Subj : S \to \mathcal{F}(2^{\mathbf{P}}, 2^{\mathbf{P}}) : Subj(s)(X) \subseteq \mathbf{B_s}$ *and*
$Subj(s)(X) = Subj(s)(X \cap \mathbf{K_s})$ *and*
$X \subseteq Y \subseteq \mathbf{P} \Rightarrow Subj(s)(X) \subseteq Subj(s)(Y)$ *(monotonic)*
$Sys : 2^{\mathbf{P}} \to 2^{\mathbf{P}} : Sys(X) \subseteq \mathbf{K}$ *and*
$Sys(X) = Sys(X \cap (\mathbf{B} \cup \mathbf{K}))$ *and*
$X \subseteq Y \subseteq \mathbf{P} \Rightarrow sys(X) \subseteq Sys(Y)$ *(monotonic)*

For every subject $s$, $Subj(s)$ is a monotonic function that represents the combined effects of the subject rules for $s$. $Sys$ is a monotonic function that represents the combined effects of the system rules. By defining them as functions from and to the set of subsets of the propagation predicates in $\mathbf{P}$, we can easily derive fix point functions from them.

**Definition 2 ($\mathcal{E}$).** *Evolution in a KBM :*
*Let* $G = < S, K, B, ar, Subj, Sys >$ *be a KBM.*
*The evolution function* $\mathcal{E}$ *of* $H$ *is :*
$\mathcal{E} : 2^{\mathbf{P}} \to 2^{\mathbf{P}} : \mathcal{E}(X) = X \cup Sys(X) \cup \bigcup_{s \in S} Subj(s)(X)$

**Lemma 1.** $\mathcal{E}$ *is increasing for* $\subseteq$.

**Definition 3 ($\mathcal{C}$).** *Completion :*
*Let* $G = < S, K, B, ar, Subj, Sys >$ *be a KBM.*
*The completion function* $\mathcal{C}$ *of* $H$ *is :*
$\mathcal{C} : 2^{\mathbf{P}} \rightarrow 2^{\mathbf{P}} : \mathcal{C}(X) = \lim_{n\rightarrow\infty} \mathcal{E}^n(X)$

**Lemma 2.** $\mathcal{C}$ *is increasing for* $\subseteq$.

**Lemma 3.** $\mathcal{C}(X) = \bigcup_{n\in\mathbb{N}} \mathcal{E}^n(X)$

These lemmas follow directly from the definitions and the monotonicity of $Sys$ and $Subj(s)$. To indicate the KBM $G$ these functions correspond to, we index them by $G : Subj_G(s), Sys_G, \mathcal{E}_G, \mathcal{C}_G$. We will denote the set of propagation predicates in the same way: $\mathbf{P}_G$.

**Theorem 1.** *Computability of finite KBM's :*
*If* $S$ *is finite and* $G = < S, K, B, ar, Subj, Sys >$, *then* $\mathcal{C}_G$ *is computable.*

**Definition 4 (Ag).** *Aggregation :*
*Let* $G = < S, K, B, ar, Subj, Sys >$ *is a KBM.*
*Let* $Ag : S \rightarrow S'$ *be a surjection.*
*By abuse of notation, let* $Ag : \mathbf{P} \rightarrow \mathbf{P'} : Ag(q(s_1, \ldots s_{ar(q)})) = q(Ag(s_1), \ldots Ag(s_{ar(q)}))$
*define* $H = < S', K, B, ar, subj', Sys' > = Ag(G)$ *such that :*
$Subj' : 2^{\mathbf{P}'} \rightarrow 2^{\mathbf{P}'} : Subj'(s')(X') = Ag(\bigcup_{s\in Ag^{-1}(s)} Subj(s)(Ag^{-1}(X')))$
$Sys' : 2^{\mathbf{P}'} \rightarrow 2^{\mathbf{P}'} : Sys'(X') = Ag(\bigcup_{s\in Ag^{-1}(s)} Sys(Ag^{-1}(X')))$

From the definition of aggregation and the lemmata, it is straightforward to check:

**Theorem 2.** *Safe approximation by aggregation :*
*Let* $G$ *and* $H$ *be KBM's and* $Ag$ *an aggregation such that* $Ag(G) = H$.
$\forall X \subseteq \mathbf{P}_G : Ag^{-1}(\mathcal{C}_H(Ag(X))) \supseteq \mathcal{C}_{\mathcal{G}}(X)$

**Corollary 1.** *Refining safe approximations by aggregation*
*Let* $G$ *and* $H$ *be KBM's and* $Ag$ *an aggregation such that* $Ag(G) = H$.
*Let* $K$ *be a KBM and* $Ag_1, Ag_2$ *aggregations :* $Ag_1(G) = K$ *and* $Ag_2(K) = H$.
$\forall X \subseteq \mathbf{P}_G : Ag^{-1}(\mathcal{C}_H(Ag(X))) \supseteq Ag_1^{-1}(\mathcal{C}_K(Ag_1(X))) \supseteq \mathcal{C}_{\mathcal{G}}(X)$

**Corollary 2.** *Computable and iteratively refined safe approximation :*
*Every KBM can be safely approximated by a series of computable and progressively more fine grained aggregations.*

Theorem 2 and corollary 2 are the main results of this formal interlude. Aggregation allows us to partition the modeled entities into a limited finite set of subject, as is most fit for the problem at hand. If a chosen aggregation turns out to be too coarse, the aggregated subject can be split up into multiple less-aggregated subjects, and the fix point of evolution can be calculated all over.

As a result of aggregation, child subjects may have more than one parent, subjects can be their own parent, and subjects can be each others child. That may seem strange at first, but it is perfectly safe, and it does not pose any problems for the theoretical model or the tool that computes the completion.

## 4 SCOLL

The language we use to describe a pattern (configuration) of collaborating subjects to be analyzed for propagation by collaboration (and creation) is a simple, structured subset of Datalog. The structure contains the system rules, subject rules, an a pattern description.

The syntax is described in EBNF format:

$$
\begin{aligned}
\langle \text{Program} \rangle &::= \langle \text{System} \rangle \, \langle \text{behaviours} \rangle \, \langle \text{Subjects} \rangle \, \langle \text{Configuration} \rangle \, \langle \text{Goals} \rangle \\
\langle \text{System} \rangle &::= \texttt{system} \, \langle \text{Rule} \rangle^{+} \\
\langle \text{behaviours} \rangle &::= \texttt{behaviour} \, \langle \text{behaviour} \rangle^{+} \\
\langle \text{Subjects} \rangle &::= \texttt{subject} \, \langle \text{Subject} \rangle^{+} \\
\langle \text{Configuration} \rangle &::= \texttt{config} \, \langle \text{Fact} \rangle^{*} \\
\langle \text{Goals} \rangle &::= \texttt{goal} \, (\langle \text{Safety} \rangle \mid \langle \text{Liveness} \rangle)^{*} \\
\langle \text{behaviour} \rangle &::= \langle \text{behaviourID} \rangle \text{ “\{” } \langle \text{Rule} \rangle^{*} \text{ “\}”} \\
\langle \text{Rule} \rangle &::= \langle \text{Pred} \rangle^{*} \text{ “=>” } \langle \text{Pred} \rangle^{+} \text{ “;”} \\
\langle \text{Pred} \rangle &::= \langle \text{PredLabel} \rangle \text{ “(” } \langle \text{VarID} \rangle^{*} \text{ “)”} \\
\langle \text{Subject} \rangle &::= (\texttt{search})^{?} \, \langle \text{SubjectID} \rangle \text{ “:” } \langle \text{behaviourID} \rangle \text{ “\{” } \langle \text{Fact} \rangle^{*} \text{ “\}”} \\
\langle \text{Fact} \rangle &::= \langle \text{PredLabel} \rangle \text{ “(” } \langle \text{SubjectID} \rangle^{*} \text{ “)”} \\
\langle \text{Safety} \rangle &::= \text{“!” } \langle \text{Fact} \rangle \\
\langle \text{Liveness} \rangle &::= \langle \text{Fact} \rangle \\
\langle \text{BehaviourID} \rangle &::= [\text{“A” } - \text{ “Z” “\_”}]^{+} \\
\langle \text{VarID} \rangle &::= [\text{“A” } - \text{ “Z”}] \, [\text{“a” } - \text{ “z” “A” } - \text{ “Z” “\_”}]^{*} \\
\langle \text{PredLabel} \rangle &::= [\text{“a” } - \text{ “z”}] \, [\text{“a” } - \text{ “z” “A” } - \text{ “Z” “\_”}]^{*} \\
\langle \text{SubjectID} \rangle &::= [\text{“a” } - \text{ “z” “\_”}]^{+}
\end{aligned}
$$

The reserved words : `system, behaviour, subject, config, goal,` and `search` cannot be used as ⟨PredLabel⟩ or ⟨VarId⟩.

### 4.1 Example : The Caretaker Pattern

In this example, the subjects *Alice*, *Caretaker*, and *Carol* are trusted while *Bob* and *Dave* are not. Untrusted subjects are given maximal behaviour for safety. *Bob* should only be able to use *Carol*'s services via *Caretaker*, a proxy to *Carol* who is created by *Alice* to give *Bob revokable* , indirect access to *Carol*.

We want to calculate the necessary restrictions (if any) for *Carol*, to make sure that *Bob* cannot get direct access to *Carol*. If it would turn out that there are no restrictions, the role of *Carol* can be played by untrusted subjects.

```
system
  iEmit(A B X) access(A B) access(A X) rCollect(B)
    => access(B X) iEmitted(A B X) rCollected(B X);
  iCollect(A B ) access(A B) rEmit(B X) access(B X)
    => access(A X) iCollected(A B X) rEmitted(B X);
  iEmitted(A B X) access(B Y) rExch(B X Y) iCollect(A B)
    => access(A Y) iExchd(A B X Y) rExchd(B X Y);
  pCreate(A B) child(A B) => access(A B) pCreated(A B);
  pEndow(A B X) access(A X) pCreated(A B)
    => access(B X) pEndowed(A B X) cEndowed(B X);
behaviour
  MAXIMAL { => iEmit(X Y) iCollect(X) rEmit(X) rCollect() rExch(X Y);}
  ALICE {
    isCT(X) isBob(B) isCarol(C)=> pCreate(X) pEndow(X,C) iEmit(B X);
    use(X) => iCollect(X) ;  use(X) pass(Y) => iEmit(X Y);
    pass(X) => rEmit(X) ;  iCollected(X Y) => use(Y);
    => rCollect() ;  rCollected(X) => pass(X);  isCarol(C) => use(C);
  CARETAKER {
     => rCollect();  IsProxy(C) => iCollect(C);
    IsProxy(C) rCollected(X) => iEmit(C X);  cEndowed(C) => isProxy(C);
    iCollected(C, X) => rEmit(X);  iExchd(C X Y) => rExch(X Y);}
  MINIMAL {}
subject
  alice : ALICE {isBob(bob) isCT(caretaker) isCarol(carol) use(alice)
                 pass(alice)}
  bob : MAXIMAL {}
  search carol : MINIMAL {}
  caretaker : CARETAKER {}
  dave : MAXIMAL {}
config
  access(alice alice) access(alice bob) access(alice carol)
  access(bob bob) access(carol carol) access(carol dave)
  access(dave dave) child(alice caretaker)
goal   access(bob dave) !access(bob carol)
```

### 4.2 Operational Semantics

We define a SCOLL program's operational semantics as a KBM. We give here an informal account of how the different parts of a SCOLL program add to the definition of the KBM $< S, K, B, ar, Subj, Sys >$ .

$S$ is the set of ⟨SubjectID⟩'s used in ⟨Subjects⟩. For the program to be valid, all instances of ⟨SubjectID⟩ in the program should be defined (mentioned) in a ⟨Subject⟩ part within ⟨Subjects⟩.

$K$ is the set of predicate labels defined in table 2. $B$ is the set of predicate labels defined in table 1. The function $ar$ defines the arity for these predicates, as in tables 1 and 2.

The function $Sys$ is defined by the rules in ⟨System⟩ and by the facts in ⟨Configuration⟩. To be valid, the rules in ⟨System⟩ should have no predicates from $B$ in their right part. The predicates in the system rules that are not in $B \cup K$ are system knowledge predicates (e.g. $access()$ and $child()$ in the example of Section 2.4), and so are the predicates in the ⟨Configuration⟩ facts.

The value of $Sys$ for a certain set of facts $X$ is calculated from the fix point of the iterative application of the system rules to the predicates in $X \cup$ ⟨Configuration⟩.

The functions $Subj(s)$ for every subject $s \in S$ are defined by the ⟨Subject⟩ part with the corresponding ⟨SubjectID⟩. To be valid, the rules in the ⟨behaviour⟩ part with the corresponding ⟨behaviourID⟩ can contain no predicates from $B$ in their left part, and no predicates from $K$ in their right part. The predicates in the subject rules that are not in $B \cup K$ are private knowledge predicates for the subject (e.g. $trust()$ and $secret()$ in the example of Section 2.3), and so are the predicates in the subject's initialization facts.

The value of $Subj(s)$ for a certain set of facts (grounded predicates) $X$ is calculated from the fix point of the iterative application of these subject rules to the predicates in $X \cup \{$initialization facts for $s\}$.

# 5 SCOLLAR

The SCOLLAR tool does more that just calculating the completion (maximal evolution) of a given SCOLL pattern. It uses techniques from concurrent constraint programming (CCP) [Sar93] to search for (all) safe combinations of behaviour in the search subjects. It makes the distinction between safe behaviour that can or can not influence the propagation (some behaviour of the search-subject might never be used because of a lack of suitable collaborators for that behaviour in the pattern).

## 5.1 Implementations

We are pursuing two complementary CCP based approaches for the implementation of SCOLLAR. CCP involves a shared store of basic constraints, that is updated by concurrent threads called *constraint propagators*. Constraint propagators can only *add* more basic constraints to the store, never remove them. This monotonic mechanism fits our purposes perfectly: we can use constraint propagators to implement both the subject rules and the system rules.

In the first approach we use *finite domain integer* (FD) basic constraints, that constrain a logic variable to a finite set of possible integer values. FD variables represent the subjects by their (integer) ID. Predicates are represented as structures of the form $q(A, B, ...D)$ where $A, \ldots D$ are FD variables representing the subject IDs. CCP in Oz [Sch02] allows the encapsulation of a (copy of the)

store in *computation spaces*. A computation space will *fail* when its copy of the store contains inconsistent basic constraints, and *succceed* when all the FD variables are completely narrowed down to one integer and no running constraint propagators can cause failure. By tuning the *search and distribution* mechanisms that govern the exploration of the search space, we are able to find all succeeding spaces while only exploring a tiny portion of the theoretical possibilities. The stores in the succeeded spaces correspond to a safe behaviour for the search subjects.

The actual implementation involves many optimisations like search heuristics to improve performance, and by-need (lazy) construction of the logic variables, to minimise memory consumption. Details on the implementation in Mozart-Oz [Moz03] of both approaches are presented in depth in a technical report [SJV05]. Since that report, we have extended the implementation to handle creation and endowment.

In the second approach we use finite sets of integers (FS) for basic constraints. These sets represent the predicate relations as such, and every grounded fact of the predicate is represented by a number.

### 5.2 Results of the example

When we feed the SCOLL code of Section 4.1 into SCOLLAR, the results show that *Carol* indeed needs to be restricted. This means that the caretaker pattern is not a safe way to implement revocable authority to arbitrary subjects. While the pattern is not suitable for general use, it is safe if *Carol*'s role is played by a subject who's behaviour is restricted in a particular way.

The tool calculates 5 alternative ways to restrict *Carol*'s behaviour in less than 5 seconds. Of the $2^{40}$ theoretical possibilities only 670 needed to be investigated. The alternatives are expressed here as behaviour *Carol* should *not* have:

1. rEmit(carol), iEmit(alice, carol), iEmit(bob, carol), iEmit(dave, carol), rExch(alice, carol), rExch(bob, carol), rExch(caretaker carol), rExch(dave, carol)
2. rEmit(alice), rEmit(carol), iEmit(alice, carol), iEmit bob, alice), iEmit(bob, carol), iEmit(dave alice), iEmit(dave,carol), rExch(bob, carol), rExch(caretaker, carol), rExch(dave,carol)
3. rEmit(alice), rEmit(carol), iEmit(bob, alice), iEmit(bob, carol), iEmit(dave, alice), iEmit(dave, carol), rExch(bob, alice), rExch(bob, carol), rExch(caretaker, alice), rExch(caretaker, carol), rExch(dave, alice), rExch(dave, carol)
4. rEmit(carol), iEmit(bob, carol), iEmit(dave, carol), rCollect()
5. rEmit(carol), iEmit(dave, carol), iCollec(dave), rCollect()

## 6 Related Research and Future Work

Except for our previous research in this field [SJV05,SV05b,SMRS04], we have not found any recent work that focusses on formal safety analysis of systems that model the propagation of authority via collaboration.

Ideas from abstraction carrying code [HALGP05] and model carrying code [SRRS01] have inspired us to investigate the use of SCOLL and SCOLLAR in this way. We want to provide support for the semi-automatic extraction of SCOLL patterns from code, and for automated checking of code compliancy with a given SCOLL pattern. A code provider could add a SCOLL pattern to his code, to reveal the authority-propagation aspects of its internals. The consumer of the code could check if the code complies to the given pattern. If compliant, the SCOLL pattern can be connected to the pattern representing the consumer code to interact with. SCOLLAR can then be used to test if the required safety properties are respected, and if not, to propose changes in the trusted entities or add some well-restricted proxy entities between the interacting parties.

To augment the expressive power of the model and the tool, we will directly support the propagation of lists of subjects. To enhance the usability, we will provide automated support for iterative refinement of patterns.

We are working on an online version of the tool for demonstration purposes.

**Conclusion** In this paper we presented a new language SCOLL, a new tool SCOLLAR, and a new formalism KBM, to describe, compute, and theoretically found the analysis of the boundaries of authority propagation for systems that allow the propagation of authority by collaboration. The approach is generally useful, but most directly applicable to systems that provide unforgeable references and avoid ambient authority.

Relying only on language runtime restrictions (e.g. capability safe languages) the behaviour of trusted entities can implement security policies. The approach and the tool allow additional or alternative safety mechanism to be modeled, including mechanisms based on Access Control by runtime reference monitoring.

Further development of this approach into a production ready development aid will allow software providers and developer to take more and well-defined responsibility (including legal responsibility) for the security of their code.

## 7 Acknowledgments

This work was funded by the EVERGROW project in the sixth Framework Programme of the European Union under contract number 001935, and by the MILOS project of the Walloon Region of Belgium under convention 114856.

## 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.

[HALGP05] Manuel V. Hermenegildo, Elvira Albert, Pedro López-García, and Germán Puebla. Abstraction carrying code and resource-awareness. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of*

*declarative programming*, pages 1–11, New York, NY, USA, 2005. ACM Press.

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

[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`.

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

[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.

[SJV05] Fred Spiessens, Yves Jaradin, and Peter Van Roy. Using constraints to analyze and generate safe capability patterns. Research Report INFO-2005-11, Département d'Ingénierie Informatique, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2005.

[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.

[SRRS01] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Model-carrying code (mcc): a new paradigm for mobile-code security. In *NSPW '01: Proceedings of the 2001 workshop on New security paradigms*, pages 23–30, New York, NY, USA, 2001. ACM Press.

[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. In *TGC 2005*, volume 3705 of *Lecture Notes in Computer Science*, pages 248–278, Berlin, Heidelberg, 2005. Springer-Verlag. Presentation available at `http://www.info.ucl.ac.be/people/fsp/auredsysfinal.mov`.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for java. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 116–128. ACM Press, 1997.