Static conflict detection for a policy language*

Alix Trieu[1], Robert Dockins[2], Andrew Tolmach[3]

1: ENS Rennes,
alix.trieu@ens-rennes.fr
2: Galois, Inc.
robdockins@fastmail.fm
3: Portland State University,
tolmach@pdx.edu

Abstract

We present a static control flow analysis used in the Simple Unified Policy Programming Language (Suppl) compiler to detect internally inconsistent policies. For example, an access control policy can decide to both "allow" and "deny" access for a user; such an inconsistency is called a *conflict*. Policies in Suppl follow the Event-Condition-Action paradigm; predicates are used to model conditions and event handlers are written in an imperative way. The analysis is twofold; it first computes a superset of all conflicts by looking for a combination of actions in the event handlers that might violate a user-supplied definition of conflicts. SMT solvers are then used to try to rule out the combinations that cannot possibly be executed. The analysis is formally proven sound in Coq in the sense that no actual conflict will be ruled out by the SMT solvers. Finally, we explain how we try to show the user what causes the conflicts, to make them easier to solve.

Introduction

Many programs are used in domains where human lives are involved: flight control software, pacemakers or for controlling nuclear plants for example. It is thus of the utmost importance to make sure that these programs behave *correctly*. Static analysis is a powerful technique to detect mistakes in programs and create robust systems. It overapproximates the set of all possible executions of a program to make sure that there cannot be a problem.

SMT (Satisfiability Modulo Theories) solvers are automated tools that decide the satisfiability of first-order formulae with respect to combinations of first-order theories (also named background theories) such as linear arithmetic or the theory of uninterpreted functions. It can sometimes take a long time to decide the satisfiability of a formula, but thanks to recent technological advances, these tools have become more powerful. They have thus gained much use in the static analysis domain to verify software (e.g., [2]).

Policies are a form of program used in many computer systems. Many policies follow the Event-Condition-Action (ECA) paradigm, and are stated in the form of **On** *Event* **If** *Condition* **Do** *Action*. Their use varies from mundane domains such as access control to critical systems such as determining what to do when an engine stops working during an airplane flight. However, policy languages are often specialized for one domain and lack the structure to allow easy static analysis. The Simple

*This work was conducted primarily while Trieu and Dockins were at Portland State University. Dockins and Tolmach were supported in part by the US Air Force Research Laboratory under contract FA8650-11-C-7189. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the funding agency.

Unified Policy Programming Language (Suppl) [9], presented in Section 1, is an attempt to address these problems. It is a language designed to allow easy combination of policies; however, this can also lead to creating inconsistencies such as deciding to both "allow" and "deny" entry at the same time for an access control policy.

Such inconsistencies can lead to fatal errors in critical systems; thus, this paper presents a static control flow analysis that makes use of SMT solvers to detect such inconsistencies. In order to improve our confidence in this analysis, it has also been formally proven sound in Coq.

1. Suppl by example

Suppl is designed to state policies following the ECA paradigm; events and actions are thus primitive concepts in the language. It also attempts to bridge two worlds: the world of logic programming through the use of predicates *à la* Prolog [8] to model conditions, and the imperative paradigm used to describe event handling in a natural way.

To explain how a policy is written in Suppl, we will examine an example. Suppose we are an e-mail service provider and want to write a simple spam filter policy. We can model this problem in Suppl using the following declarations:

```
type address := string.
type user := string.
predicate trusted(address).
trusted("bob@leponge.com").
```

We define types `address` and `user` to be strings and a predicate named `trusted` which uses an `address` as argument. We then indicate that `bob@leponge.com` is a trusted address.

```
event receive(address, user).
data category ::= inbox | spam.
action send(category, user).
```

Here, we define an event named `receive` to indicate that the address given as first argument is attempting to send an e-mail to the user given as second argument. We also define a data type named `category`, which can either be `inbox` or `spam`. Finally, we define the action `send`, which is used to decide where the e-mail should be sent.

```
handle receive(?A, ?U) =>
  query
  | trusted(A) => send(inbox, U);
  end;
end.
```

Here, we define an event handler saying that if the sender's address is trusted, then the e-mail should be sent to the inbox of the recipient. Otherwise, no action is taken.

Now, to make this policy a little bit more interesting, we need to add some properties to the system. For example, we may want to record complaints from our users concerning some addresses.

```
table unwanted(address, user) key (in, in).

predicate isSpam(address).
isSpam("carlo@tentacule.com").
```

```
isSpam(A) :-
  findall(U, unwanted(A, ?U), RS), set_size(RS) >= 10.
```

The only way in SUPPL to keep track of a state is to use data tables to record this information. In the logic part of the language, tables are considered like predicates, but the imperative event handling part provides commands to insert and delete rows from tables. We first define a table `unwanted` to record the addresses the users consider to be sending them spam. The table has two columns with types `address` and `user`. The `key` keyword is used to define the table's primary key, the `in` and `out` keywords following `key` indicates which columns form the table's primary key: columns declared with `in` are in the primary key, those declared with `out` are not. Every table will contain at most one row for the values in the primary key. If a new row is inserted with the same values for all primary key columns as a row already in the table, the old row will be evicted and the new row will replace it. Therefore, we are indicating for this table that there cannot be two rows with the same exact values. We then say that an address `A` is considered a spammer if we manage to find complaints from at least 10 different users involving the address `A`. This is done by asking to find all the `U` such that `unwanted(A, U)` holds and putting them into the set `RS`.

```
handle receive(?A, ?U) =>
  query
  | isSpam(A) => send(spam, U);
  | unwanted(A, U) => send(spam, U);
  end;
end.
```

Here we add a second event handler: if the address is not trusted then its message is sent to the spam box of the recipient; moreover, if the address is not considered a spammer, but this user doesn't want messages from it, then the message is also forwarded to the spam box. A query is similar to a *switch* statement in imperative programming; the body corresponding to the first query clause that holds true is executed. When the event `receive` is raised, *both* handlers are executed, and a set of actions to execute is thus computed.

Obviously, this example is quite simple and doesn't present the more complex features of SUPPL. In particular, the language allows the use of arbitrary recursive predicate, which makes the language Turing-complete. The language is more exhaustively presented in [9] and the development is available at [10].

2. Conflict detection

SUPPL is designed so that combining code from different sources is easy. However, this also makes it possible to write self-contradictory policies. In the previous example, it is possible for the system to decide that the received message must be transmitted to both the inbox and the spam box of the recipient. We call such contradiction a *conflict*. Our solution to this problem is to develop a tool that can *statically* detect potential conflicts. However, it is not obvious how to automatically determine what constitutes a conflict. Therefore, the user needs to supply a *conflict declaration* such as

```
conflict send(?A, ?U), send(?B, ?V) => A <> B, U = V.
```

which explains to the system that the set of actions resulting from an event must not contain the actions `send(A, U)` and `send(B, V)` such that `A` and `B` are not equal and `U = V`. This means that a policy deciding to send an e-mail to both the inbox and spam box of the same user is incoherent.

Specifically, our analysis identifies control-flow paths through a policy that are initiated from the same event but lead to conflicting actions. For each pair of control-flow paths that lead to conflicting

$b :=$		handler body
	$act(a_1, \ldots, a_n)$	action
	`skip`	skip
	b_1 `;` b_2	sequence
	`foreach` $p(a_1, \ldots, a_n)$ `=>` b `end`	foreach
	`query` $brs_1; \ldots; brs_n$ `end`	query
$brs :=$		query branches
	`branch` qcl b	query branch
	`defaultbranch` b	default branch
$qcl :=$		query clauses
	$a_1 = a_2$	term matching
	qcl_1 `,` qcl_2	conjunction
	$p(a_1, \ldots, a_n)$	predicates
$cdecl :=$		conflict declaration
	`conflict` $act_1(a_1, \ldots, a_n)$ `,` $act_2(b_1, \ldots, a_m)$ `=>` qcl `.`	with a clause
	`conflict` $act_1(a_1, \ldots, a_n)$ `,` $act_2(b_1, \ldots, a_m)$ `.`	without a clause

Figure 1: Abstract syntax of event handlers and conflict declarations

actions, we then generate a formula in first-order logic that states what conditions would have to be true for the program to follow both these paths on a single event occurrence. In a second phase, we pass the generated formulae to SMT solvers such as Alt-Ergo [6] or CVC4 [3] through Why3 [12], which is a platform that uses multiple external solvers. If the solver can show that a formula is unsatisfiable, then we know that the corresponding potential conflict cannot actually occur. Otherwise, we report the potential conflict to the user.

In the context of safety-critical software, it is of utmost importance to make sure that the software contains no bugs. Formal verification is such a way to provide guarantees about the specification and implementation of a program. It is done by using machine-checked proofs, which are more reliable than manual verification. Thus, we formally prove the analysis to be sound for a core subset of the language, in the sense that the conflict detection scheme reports a potential conflict for each actual conflict; in other words, a policy that passes the analysis without complaints is guaranteed to run without any conflicts.

2.1. Syntax

We detail in Fig. 1 only a core subset of Suppl related to event handler bodies, as this is the only part that matters in our conflict detection scheme. (In the code examples in this paper, we use a slightly different syntax for queries, in which query branches are separated by vertical bars (`|`), `branch` keywords are omitted, and the `defaultbranch` keyword is written as underscore (`_`)). As explained earlier, a handler body may contain commands to insert or delete rows from a table. As they are not involved in the conflict detection, these commands are not included in Fig. 1, but remember that in the logic part, tables are considered predicates.
We will use act as an identifier for actions, t as an identifier for tables, p as an identifier for predicates and $a_1, \ldots, a_n, b_1, \ldots, b_m$ as ground terms, already bound variables or binding variables. Binding variables are preceded by a question mark `?`; this means that the variable is not currently bound to any ground term in the current environment. They are used in cases such as `foreach` p(3, `?`a) `=>` b `end` to say "find all values x such that p(3, x) holds and execute b for each of these values x with $a = x$".

2.2. Semantics

The language has been informally explained in Section 1; we now give a formal semantics for an event handler's execution. Our goal is to describe when multiple actions can occur in response to a single event.

1. A query is similar to a switch statement from imperative programming: only one branch is ever executed at runtime, thus the set of actions reachable in a query comes only from one of its branches.
2. As its name indicates, in a `foreach`, the body is executed for each combination of values that makes the predicate hold true, thus the actions come from each iteration of the body.
3. In the case of a sequence, the actions from the first and second body are both executed.

We suppose hereafter that we have only one handler. This assumption is valid because when there are multiple handlers for the same event, the actions of all handlers are executed; thus, it is actually similar to the case of a sequence. Let σ be an environment (mapping from variables to ground terms). We define in Fig. 2 a nondeterministic relation $b \xrightarrow{\sigma} \ell$ where b is a body and ℓ is a list of pairs of environments and actions. $(\sigma', a) \in \ell$ means that there exists an execution of the body b in the environment σ such that the action a is executed in the environment σ'. This definition is nondeterministic because of the way predicates are queried. For example, if we have a predicate `p` such that only `p(1)` and `p(2)` are true, there are two possible executions of `query | p(?N) => b; end;` one where `N` is bound to `1` in b, the other one where `N` is bound to `2`.

The nondeterministic relation $\sigma \underset{qcl}{\longrightarrow} \sigma'$, defined in Fig. 3, says that σ' is an extension of the environment σ (written $\sigma \triangleright \sigma'$) such that the query clause qcl holds in σ'. Note that the semantics given in Fig. 2 gives an approximation of the actual behaviors of an event handler in the real SUPPL language, in the sense that it permits more executions. In particular, it allows `foreach` statements to terminate at any point, without necessarily exhausting all possible instantiations of the controlling query; this inaccuracy is harmless for our purposes, because any conflict that can arise in the real language will still appear under this semantics.

2.3. Definition of conflicts

Definition 1 (actual conflict)**.** An *actual* conflict $(\sigma_1, a_1(x_1, \ldots, x_n)) \bowtie_c (\sigma_2, a_2(y_1, \ldots, y_m))$ between two (environment, action) pairs occurs when c is a *conflict declaration* `conflict` $a_1(z_1, \ldots, z_n)$, $a_2(t_1, \ldots, t_m)$ `=>` cl such that the clause cl holds when all z_k are substituted by $\sigma_1(x_k)$ and all t_k are substituted by $\sigma_2(y_k)$.

The goal of our analysis is to discover all *actual* conflicts of a policy. However, because of the static nature of our analysis, there must be a tradeoff, which is the completeness of our analysis. Indeed, our analysis can report false positives. In order to reduce the rate of false positives, we introduce a data structure named PConflict to represent *potential* conflicts.

Definition 2 (PConflict)**.** A *PConflict* $(ev(z_1, \ldots, z_k), cp, d_1, d_2, a_1(x_1, \ldots, x_n), a_2(y_1, \ldots, y_m), \ell)$ consists of an event (ev) producing the potential conflicts, the arguments $(z_1, \ldots, z_k)$ of the event, control flow paths (cp, d_1, d_2) leading to the two conflicting actions, the two conflicting actions $(a_1(x_1, \ldots, x_n), a_2(y_1, \ldots, y_m))$ themselves and the conflict declarations (ℓ) that they might satisfy.

Definition 3 (Control Flow Path)**.** A control flow path (Fig. 4) is represented by a list of query clauses (as defined in Section 2.1). The clauses are either positive or negative (preceded by a $\neg$). A positive clause must hold for the action, at the end of the path, to be executed; conversely, for a negative clause $\neg qcl$, qcl must not hold for the action to be executed.

$$\frac{}{act(a_1, \ldots, a_n) \xrightarrow{\sigma} \texttt{[}(\sigma, act(a_1, \ldots, a_n))\texttt{]}} \text{ (actions)}$$

$$\frac{b_1 \xrightarrow{\sigma} \ell_1 \qquad b_2 \xrightarrow{\sigma} \ell_2}{b_1; b_2 \xrightarrow{\sigma} \ell_1 \texttt{++} \ell_2} \text{ (seq)}$$

$$\frac{}{(\texttt{foreach}\ qcl\ b) \xrightarrow{\sigma} \texttt{[]}} \text{ (foreachN)}$$

$$\frac{(\texttt{foreach}\ qcl\ b) \xrightarrow{\sigma} \ell \quad \sigma \underset{qcl}{\longrightarrow} \sigma' \quad b \xrightarrow{\sigma'} \ell'}{(\texttt{foreach}\ qcl\ b) \xrightarrow{\sigma} \ell \texttt{++} \ell'} \text{ (foreachS)}$$

$$\frac{}{(\texttt{query}\ \texttt{[]}) \xrightarrow{\sigma} \texttt{[]}} \text{ (querynil)}$$

$$\frac{\forall \sigma', \neg\ (\sigma \underset{qcl}{\longrightarrow} \sigma') \quad (\texttt{query}\ qbrs) \xrightarrow{\sigma} \ell}{(\texttt{query}\ (\texttt{branch}\ qcl\ b) : qbrs) \xrightarrow{\sigma} \ell} \text{ (queryF)}$$

$$\frac{\sigma \underset{qcl}{\longrightarrow} \sigma' \qquad b \xrightarrow{\sigma'} \ell}{(\texttt{query}\ (\texttt{branch}\ qcl\ b) : qbrs) \xrightarrow{\sigma} \ell} \text{ (queryT)}$$

$$\frac{b \xrightarrow{\sigma} \ell}{(\texttt{query}\ (\texttt{defaultbranch}\ b) : qbrs) \xrightarrow{\sigma} \ell} \text{ (querydefault)}$$

$$\frac{}{\texttt{skip} \xrightarrow{\sigma} \texttt{[]}} \text{ (skip)}$$

Figure 2: Inference rules for $b \xrightarrow{\sigma} \ell$

$$\frac{\sigma \xrightarrow[qcl_1]{} \sigma' \quad \sigma \xrightarrow[qcl_2]{} \sigma'}{\sigma \xrightarrow[qcl_1, qcl_2]{} \sigma'} \text{ (and)}$$

$$\frac{\sigma(a_1) = \sigma(a_2)}{\sigma \xrightarrow[a_1 = a_2]{} \sigma} \text{ (matching)}$$

$$\frac{}{\sigma \xrightarrow[?a_1 = a_2]{} \sigma[a_1 \leftarrow \sigma(a_2)]} \text{ (matchingleft)}$$

$$\frac{}{\sigma \xrightarrow[a_1 = ?a_2]{} \sigma[a_2 \leftarrow \sigma(a_1)]} \text{ (matchingright)}$$

$$\frac{\sigma \triangleright \sigma' \quad p(\sigma'(a_1), \ldots, \sigma'(a_n))}{\sigma \xrightarrow[p(a_1, \ldots, a_n)]{} \sigma'} \text{ (pred)}$$

Figure 3: Inference rules for $\sigma \xrightarrow[qcl]{} \sigma'$

path :=	`[]`	empty path
	qcl:p	positive clause
	$\neg qcl$:p	negative clause

Figure 4: Syntax of paths

Definition 4 (Satisfiable)**.** A path p is *satisfiable* under an environment σ (written $\sigma \models p$) if all the positive clauses of p and the negations of the negative clauses of p hold after instantiating variables using σ.

The two paths in a PConflict may have a common prefix, which we store in *cp*. The two divergent parts are stored in d_1 and d_2. This information is vital to check whether these *potential* conflicts are *realizable* and are *actual* conflicts as explained in Section 2.6.

Definition 5 (Realizable)**.** A PConflict $(ev(z_1, \ldots, z_k), cp, d_1, d_2, a_1(x_1, \ldots, x_n), a_2(y_1, \ldots, y_m), \ell)$ is *realizable by* the actual conflict $(\sigma_1, a_1(x_1, \ldots, x_n)) \bowtie_c (\sigma_2, a_2(y_1, \ldots, y_m))$ if $c \in \ell$ and there exists an environment σ such that $\sigma_1 \models d_1$, $\sigma_2 \models d_2$, $\sigma \triangleright \sigma_1$, $\sigma \triangleright \sigma_2$, and $\sigma \models cp$. A PConflict is *realizable* if it is realizable by some actual conflict.

We will explain in more detail in Section 2.6 how we check for path satisfiability in practice.

2.4. Algorithm

We now describe an algorithm for detecting potential conflicts. The algorithm takes the program's abstract syntax tree (AST) as input and returns PConflict data structures. It follows multiple steps :

1. From the program AST, we get a list of event handlers.
2. We then group the handlers by the event they handle, thus giving us a list of lists of event handlers.

3. For each group of handlers, we construct a single *equivalent* handler. This is done by :

   (a) Looking for the handler that binds the most variables used by the event.

   (b) Renaming all variables in the handlers and their bodies so that the variables bound at the event level are the same for each handler.

   (c) Forming a single handler body from all the bodies by concatenating them in sequence.

   The idea behind this step is that a conflict that lies in two different handlers is the same as a conflict happening by executing the body of the first handler and then executing the body of the second handler (thus in sequence).

4. Finally, we look for the potential conflicts within each handler. This is roughly done by computing all paths leading to an action and checking for each pair of paths whether they conflict. This is what the *is_pconflict* function is used for in the definition of *combine*, below; it returns the list of all conflict declarations in the program that have the two actions' names. We use three functions to generate the potential conflicts, the main function *pconflicts* and auxiliary functions *paths* and *combine*. The functions have the following signatures :

   $$\begin{array}{ll} \mathit{pconflicts}: & \mathit{Path} \to \mathit{Body} \to \mathit{PConflict\ List} \\ \mathit{paths}: & \mathit{Body} \to \mathit{(Actions,\ Args,\ Path)\ List} \\ \mathit{combine}: & \mathit{Path} \to \mathit{(Actions,\ Args,\ Path)\ List} \to \\ & \mathit{(Actions,\ Args,\ Path)\ List} \to \mathit{PConflict\ List}. \end{array}$$

   Here, *Path* is a control flow path as defined in Section 2.3 and Fig. 4. *pconflicts* is given a prefix for the common path and computes all the potential conflicts that can happen within the body given as second argument.

   $$\begin{array}{l} \mathit{pconflicts\ pre\ b} = \\ \quad \text{case } b \text{ of} \\ \quad b_1\,\texttt{;}\,b_2 \to \quad \mathit{pconflicts\ pre\ } b_1 \texttt{ ++} \\ \qquad\qquad\qquad\qquad\quad \mathit{pconflicts\ pre\ } b_2 \texttt{ ++} \\ \qquad\qquad\qquad\qquad\quad \mathit{combine\ pre\ (paths\ } b_1) \ (\mathit{paths\ } b_2) \\ \quad \texttt{foreach}\ p(a_1, \ldots, a_n)\ \texttt{=>}\ b' \to \quad \mathit{pconflicts\ (}p(a_1, \ldots, a_n)\mathit{:pre)\ } b' \texttt{ ++} \\ \qquad\qquad\qquad\qquad\quad \mathit{combine\ pre} \\ \qquad\qquad\qquad\qquad\qquad (\mathit{paths\ (}\texttt{foreach}\ p(a_1, \ldots, a_n)\ \texttt{=>}\ b')) \\ \qquad\qquad\qquad\qquad\qquad (\mathit{paths\ (}\texttt{foreach}\ p(a_1, \ldots, a_n)\ \texttt{=>}\ b')) \\ \quad \texttt{query}\ (\texttt{branch}\ \mathit{qcl}\ b)\mathit{:bs} \to \quad \mathit{pconflicts\ (qcl:pre)\ } b \texttt{ ++} \\ \qquad\qquad\qquad\qquad\quad \mathit{pconflicts\ (}\neg \mathit{qcl:pre)\ (}\texttt{query}\ \mathit{bs}) \\ \quad \texttt{query}\ (\texttt{defaultbranch}\ b)\mathit{:bs} \to \quad \mathit{pconflicts\ pre\ } b \\ \quad \texttt{\_} \to \quad \texttt{[]} \end{array}$$

   In the case of a sequence, the conflicts can come from each bodies, but also from an action in the first body and another action in the second body. Similarly, in the case of `foreach`, conflicts can arise from within the same iteration of the body, but also from two different iterations of the body. In a `query`, only one branch can be executed; therefore, two different branches cannot conflict with each other.

   paths computes the list of possibly reached actions in the body given as argument and the control flow paths used to reach the actions.

```
paths b =
  case b of
  b;b' →                          paths b ++ paths b'
  foreach q(a_1,...,a_n) => b →   map (λ(act, args, p) ↦ (act, args, q(a_1,...,a_n):p)) (paths b)
  query (branch qcl b):qbrs →     map (λ(act, args, p)↦ (act, args, qcl:p)) (paths b) ++
                                  map (λ(act, args, p)↦ (act, args, ¬qcl:p)) (paths (query qbrs))
  query (defaultbranch b):qbrs →  paths b
  query [] →                      []
  act(a_1,...,a_n) →              [(act, (a_1,...,a_n), [])]
  _ →                             []
```

combine computes the potential conflicts given the paths received as arguments. This is done by getting all possible pairs of path/actions and checking if a conflict is possible.

```
combine pre ℓ ℓ' =  concat
                      map (λ(act, args, p)↦
                        (concat
                          map (λ(act', args', p') ↦
                            case is_pconflict(act, act') of
                              [] → []
                              c → [(pre, p, p', act, args, act', args', c)]) ℓ')) ℓ
```

2.5. Formal proof

In order to prove the soundness of our conflict detection scheme, we need to show that for each actual conflict caused by actions that can be executed, our algorithm will generate a corresponding *realizable* PConflict data structure (as defined in Section 2.3). More precisely, if we write σ_0 for the empty environment, we have:

Soundness theorem: Suppose $b \xrightarrow{\sigma_0} w$ and w contains two entries $(\sigma_1, a_1(args_1))$ and $(\sigma_2, a_2(args_2))$ such that $(\sigma_1, a_1(args_1)) \bowtie_c (\sigma_2, a_2(args_2))$. Then *pconflicts* `[]` b contains a PConflict $(cp, d_1, d_2, a_1(args_1), a_2(args_2), \ell)$ that is realizable by $(\sigma_1, a_1(args_1)) \bowtie_c (\sigma_2, a_2(args_2))$.

Before proving the theorem, we first state the following lemma, which says that for all actions resulting of the execution of the body b, there exists a path leading to the action inside of b that is computed by the function *paths*.

Lemma: Suppose $b \xrightarrow{\sigma} w$. Then for each environment $(\sigma', a) \in w$, there exists a path p such that $\sigma' \models p$ and $(a, p) \in paths\ b$.

Proof of lemma:
The proof is by structural induction of the derivation of $\xrightarrow{\sigma}$.

Proof of theorem:
By structural induction on the derivation of $\xrightarrow{\sigma}$.

1. If b is an action; then $a_1(arg_1) = a_2(arg_2)$, thus there is no conflict and the theorem holds.

2. If $b = b_1;b_2$, by definition, there exist u and v such that $w = u{+}{+}v$ and $b_1 \xrightarrow{\sigma} u$ and $b_2 \xrightarrow{\sigma} v$. If $(\sigma_1, a_1(arg_1))$ and $(\sigma_2, a_2(arg_2))$ are both in u or both in v, we only need to use the induction hypothesis. Otherwise, if $(\sigma_1, a_1(arg_1)) \in u$ and $(\sigma_2, a_2(arg_2)) \in v$, then by the lemma there exist d_1 and d_2 that are suitable candidates. Furthermore, since *is_conflict* $(\sigma_1, a_1(arg_1))$ $(\sigma_2, a_2(arg_2))$ holds, we know that *is_pconflict*(a_1, arg_1, a_2, arg_2) is non-empty and there exists c such that (cp_0, d_1, d_2, c) is suitable. The proof is similar if $(\sigma_1, a_1(arg_1)) \in v$ and $(\sigma_2, a_2(arg_2)) \in u$.

3. If $b =$ `foreach` qcl `=>` b' and $w =$ `[]`, then we get a contradiction as $(\sigma_1, a_1(arg_1)) \in w$. Therefore, there exist ℓ and ℓ' such that $w = \ell{+}{+}\ell'$ and (`foreach` qcl b') $\xrightarrow{\sigma} \ell$ and $\sigma \xrightarrow[qcl]{} \sigma'$ and $b \xrightarrow{\sigma'} \ell'$. If $(\sigma_1, a_1(arg_1)) \in \ell$ and $(\sigma_2, a_2(arg_2)) \in \ell$, then we just need to use the induction

hypothesis. Otherwise, we use the lemma twice to get p_1 and p_2. If $(\sigma_1, a_1(arg_1)) \in \ell'$, we define d_1 as $qcl : p_1$, otherwise $d_1 = p_1$. Likewise for d_2. Finally, we can conclude that exists c such that (cp_0, d_1, d_2, c) is suitable.

4. For the querynil and skip rules, we have a contradiction as $w =$ `[]` and $(\sigma_1, a_1(arg_1)) \in w$.
5. For the queryF and queryT rules, we only need to use the induction hypothesis and add qcl or $\neg qcl$ to the common prefix to conclude.
6. Lastly, for the querydefault rule, we only need to use the induction hypothesis to conclude.

We can then easily prove the corollary that says that if all PConflicts are unrealizable, then there will be no actual conflict at runtime. The lemma and theorem have been verified in Coq [14] and is available at [10].

2.6. Formulae generation

Why3 is a platform for deductive program verification that relies on automated theorem provers to discharge its verification tasks. We translate PConflict data structures generated in the previous step into first-order formulae to be passed off to Why3. This is why the structures contain so much information: we need to help the solvers as much as possible so that they produce as few false positives as possible. We first build the *background* theory. This is a mostly straightforward step, as the translation to the Why3 input language is almost one-to-one. All the types, predicates, and table declarations are translated into the background theory. Since Why3 does not natively support equality over strings, they are translated as constants represented by lists of integers corresponding to their ASCII encoding. For example, here are some of the translations from Suppl to Why3:

Suppl	Why3
`type user := string`	`type user = string`
`data category ::=` `  inbox | spam`	`type category = Z_inbox | Z_spam`
`bob@leponge.com`	`constant string4 :  string =` `  (Cons 98 (Cons 111 (Cons 98 ...`

`predicate trusted(address)`
`trusted("bob@leponge.com")`
`trusted("patrick@etoilede.mer")`
	`predicate trusted (_arg0 :  (address))`
	`  = ((_arg0 = string4) \/`
	`  (_arg0 = string5))`

`table unwanted(address, user) key (in, in)`	`predicate unwanted (address) (user)`

The next part is to use the PConflict structures and verify if they are *realizable* conflicts. This is done by telling the solvers what would have to be true in order for the two actions to be conflicting and verifying that it *cannot* happen. This is effectively translating the control flow paths. For example, if we have a conflict declaration `conflict a(?N), a(?M) => N <> M` and the following body where `p` and `q` are predicates over numbers:

```
query
| p(?N) => query
          | q(N) => a(N);
          end;
          query
          | q(N) => a(N);
          end;
end;
```

We would get a PConflict where the common path is `p(?N)` and the two divergent paths are `q(N)` and `q(N)`. Thus, to have an *realizable* conflict, we need that there exists a `N` such that `p(N)`, `q(N)` and `N <> N`.

```
axiom conflict : exists _x0 : real.
                 ((p _x0) /\
                 ((q _x0) /\
                 ((q _x0) /\
                 (not (_x0 = _x0)))))
goal impossible : false
```

We ask the solvers to prove that it is actually *impossible*, which is the case here as obviously `not (_x0 = _x0)` cannot be true.

Note that to discover this impossibility, it is crucial that the *same* variable is used to model `N` in both queries. This is the purpose of maintaining a common subpath in the PConflict structure; if we just stored two distinct paths from event to actions, we would have generated a satisfiable formula:

```
axiom conflict : exists _x0 : real.
                 ((p _x0) /\
                 ((q _x0) /\
                 (exists _x1 : real.
                 ((p _x1) /\
                 ((q _x1) /\
                 (not (_x0 = _x1)))))))
goal impossible : false
```

3. Feedback to the user

The conflict detection algorithm generates *potential* conflicts that are passed off to external SMT solvers, which verify if the corresponding formulae are unsatisfiable and hence cannot *actually* happen at runtime. On the other hand, conflicts that the solvers are not able to prove unsatisfiable are reported to the user. But if the user only gets back which control-flow paths and which actions caused the conflict, this might not be sufficient to understand *why* the conflict may be happening. For example, suppose we have the two following paths from the policy described in Section 1, highlighted in red and blue, leading to a conflict that could not be ruled out.

```
handle receive(?A, ?U) =>
  query
  | trusted(A) => send(inbox, U);

  end;
end.
```

```
handle receive(?A, ?U) =>
  query
  | isSpam(A) => send(spam, U);
  | unwanted(A, U) => send(spam, U);
  end;
end.
```

The user might not immediately understand why it is possible for the address `A` to be considered both `trusted` and `isSpam`. Thus, we have implemented an experimental scheme that tries to report to the user which rules in the predicate definition may cause the conflict. This is accomplished by going through the control-flow paths of the conflicts and looking at which predicates are present in positive guards. Following this step, we generate different versions of the potential conflict by creating different backgrounds in which the involved predicates each only have one rule. These different versions of the potential conflict are then given back to the solvers. In the example, we only have two predicates involved, `trusted` and `isSpam`; `unwanted` is not involved as it doesn't

Figure 5: An example of how the results are presented to the user. The two involved rules are highlighted in dark green. The two paths are highlighted in salmon and light blue in the handlers.

appear in any of the two highlighted paths. `trusted` has only one rule and `isSpam` has two rules, so we split the original conflict in two cases. The first case is the one where `isSpam` only has the one rule `isSpam("carlo@tentacule.com").` The second case is the one where `isSpam` only has the other rule `isSpam(A) :- findall(U, unwanted(A, ?U), RS), set_size(RS) >= 10.` We then give back these two problems to the SMT solvers. The first case is easily ruled out by the solvers. However, the second case is not; indeed, a keen reader might have noticed that there is nothing to say that a trusted address cannot have more than 10 users considering it a spammer, which would cause a conflict. So this potential conflict is reported to the user. This is done through a pretty-printed HTML file to explain which predicates and specifically which rules cause the conflict; the two control flow paths leading to the conflicting actions are highlighted in salmon and light green in the handlers, while the two involved rules are highlighted in dark green as seen in Fig. 5.

However, a predicate might have many more than two rules, which would create many different versions of the same conflict. In this case, we try to help the user by checking if all rules of a predicate still appear in the different versions of the conflict that the solvers were not able to solve. If this is the case, we can deduce that whatever the rule that this predicate follows, the conflict still exists, meaning that the predicate *is not the cause* of the conflict and is thus not highlighted in the display.

```
predicate p(number).
p(N) :- N > 4.
p(N) :- N < 0.

predicate q(number, number).
q(N, M) :- N > M + 5.
q(N, M) :- M = 0.

predicate r(number, number).
r(N, M) :- M > 10.
r(N, M) :- N > M + 2.
```

```
handle ev(?N, ?M) =>
  query =>
  | p(N) => query
            | q(N, M) => act1;
            end;
  end;
end.
```

```
handle ev(?N, ?M) =>
  query =>
  | r(N, M) => act2;
  end;
end.
```

Figure 6: One possible version of the potential conflicts

In the situation of Fig. 6, neither version of the conflict with combination of rules

```
p(N) :- N > 4.
q(N, M) :- N > M + 5.
r(N, M) :- N > M + 2.
```

```
p(N) :- N < 0.
q(N, M) :- N > M + 5.
r(N, M) :- N > M + 2.
```

can be ruled out. Therefore, neither of the rules of predicate `p` is highlighted.

Note that this case differentiation is not necessarily a sound approach for all conflicts. Indeed, in some cases, all different versions of the conflict can be ruled out, but the original cannot.

```
predicate friend(string, string).
friend("Gary", "Bob").
friend("Bob, "Patrick").
friend(A, C) :- friend(A, ?B), friend(B, C).

query =>
  | friend("Gary", "Patrick") => act1;
end;
```

```
query =>
  | _ => act2;
end;
```

Indeed, in this situation, our case differentiation scheme is useless since the conflict needs all three rules for it to be realizable.

There are several ways for the user to prevent the solver from reporting a potential conflict. The SUPPL language supports the use of axioms and lemmas in the source code. For the example given in Section 1, we can add an axiom to tell the compiler that the conflict is not possible : `axiom trusted(A) -> (findall(U, unwanted(A, ?U), RS), set_size(RS) = 0).` This says that if address `A` is trusted, then no user has filed a complaint concerning this address. Note that this facility is obviously very unsafe. On the other hand, specifying a lemma will cause the solvers to try to prove the lemma first. The compiler also accepts a keyword `presume` that is used to tell the solvers that events must always follow a given clause. For example, we can say that we can only receive e-mails from `"sheldon@plankton.com"`: `presume receive(?A) => A = "sheldon@plankton.com"`.

These features can possibly lead the user to accidently create an inconsistency that would then render all conflicts unsatisfiable. Therefore, if one of these constructs is used in the policy, we first ask the SMT solvers to try to prove *false* and raise an error if they do manage to prove it. However, this is unfortunately not a complete check, since the solvers may not manage to find an inconsistency even if one exists.

Furthermore, as tables are simply treated like predicates in the logic side of the compiler, we also automatically generate axioms so that the SMT solvers can understand that there can only be one row for the values in the table's primary key. For example, if we had the following table declaration `table unwanted(address, user) key(in, out)`, we would then generate the following `axiom unwanted(A1, A2) -> unwanted(B1, B2) -> A1 = B1 -> A2 = B2`, which, in Why3, is generated as:

```
axiom axiom_table_unwanted : (forall _x4 : (user),
                                     _x3 : (address),
                                     _x2 : (user),
                                     _x1 : (address).
                                ((not (unwanted _x1 _x2)) \/
                                 ((not (unwanted _x3 _x4)) \/
                                  ((not (_x1 = _x3)) \/
                                   (_x2 = _x4)))))
```

This work is necessary to have an as faithful as possible translation of the background theory in order to decrease the number of false positives.

4. Related work

Conflict detection has been extensively studied, for example, Lupu and Sloman [13] present a tool for static detection of conflicts in a role-based management framework (policies define which roles have authorization or obligation of certain actions). Unlike SUPPL's domain-neutral approach, Ben Youssef et al. [4] propose an automatic method using the SMT solver Yices [11] to verify that there is no conflict within a security policy for firewalls. However, as far as we know, our tool is the first formally verified static conflict detection analysis.

The use of automated theorem provers is not a novel idea and they have been used for a long time in others domains such as bounded model checking [5] which is a technique that makes use of SAT solvers to automatically detect errors in finite state systems. This technique was then extended to use SMT solvers by Armando et al. [1]. Our refinement scheme where a conflict is split into multiple cases is vaguely similar to the counterexample-guided abstraction refinement (CEGAR) [7] model checking method in the sense that it refines an abstract model in order to avoid false counterexamples.

5. Conclusion

SUPPL is an attempt to create a domain-neutral language for the ECA paradigm, while keeping the possibility of having static analyses. Thus, the static control flow analysis for detecting conflicts in policies presented in Section 2 has been implemented into the compiler. Furthermore, in order to try helping users understand the causes of a conflict, the scheme presented in Section 3 has also been created.

However, the feedback to the user can still be improved. Indeed, it might also be useful to report a possible instantiation of the conflict to the user. Unfortunately, Why3 does not analyze the output of an automated prover and therefore does not report the possible counterexamples found by the prover. Moreover, the current system only supports pairwise policy conflicts while the user may need to specify *n-way* conflicts.

References

[1] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.

[2] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. PLDI*, pages 203–213, 2001.

[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proc. Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] N. Ben Youssef, A. Bouhoula, and F. Jacquemard. Automatic verification of conformance of firewall configurations to security policies. In *Proc. ISCC*, pages 526–531. IEEE, 2009.

[5] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[6] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover. `http://alt-ergo.lri.fr`, 2008.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[8] A. Colmerauer. An introduction to prolog III. In J. W. Lloyd, editor, *Computational Logic*, ESPRIT Basic Research Series, pages 37–79. Springer Berlin Heidelberg, 1990.

[9] R. Dockins and A. Tolmach. SUPPL: A flexible language for policies. In *Proc. 12th Asian Symposium on Programming Languages and Systems*, pages 176–195. Springer, 2014.

[10] R. Dockins, A. Tolmach, and A. Trieu. SUPPL. `http://web.cecs.pdx.edu/~rdockins/suppl`.

[11] B. Dutertre and L. de Moura. Yices. `http://yices.csl.sri.com/`.

[12] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *Proc. ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, Mar. 2013.

[13] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, 1999.

[14] The Coq development team. Coq. `http://coq.inria.fr/`.