ÉNS Rennes
Portland State University

Static analysis of a policy language

Internship Report

Author :
Alix Trieu

Supervisors :
Robert Dockins
Andrew Tolmach

Internship from May 19, 2014 to August 14, 2014 at
Portland State University.

Abstract. We present here a static control flow analysis used in the Simple Unified Policy Programming Language (Suppl) compiler to detect internally inconsistent policies. These inconsistencies are called *conflicts*. The analysis is twofold, it first detects *potential* conflicts and then eliminates the un*realizable* conflicts through the use of SMT solvers. Supposing that inputs to the solvers are correctly generated, the analysis is formally proven sound with the help of the proof assistant Coq.

Keywords: Automatic Theorem Proving, Policy, Conflict Detection, Static Analysis, Why3, Logic Programming, Coq, SMT

1 Introduction

Static analysis is a powerful discipline to detect mistakes in programs and create robust systems. It is extensively used in critical domains such as flight control software [9]. 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 [2]. However, their use has not developed in the domain of policy specification yet.

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 Unified Policy Programming Language (Suppl) [11], presented in section 2, 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, my main contribution is a static control flow analysis that makes use of SMT solvers to detect such inconsistencies. This analysis supersedes the previous analysis implemented in the Suppl compiler and I have formally proved its soundness; it will be introduced in section 3. Lastly, I show and explain the tools and improvements to the compiler I made to help users design sound policies in section 4.

2 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 writing a simple policy to decide whether an e-mail is considered a spam or not. 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.

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 at least 10 complaints from 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`.

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

Here, we define an event named `receive` to indicate that the address given as argument is attempting to send an e-mail to one of our users. 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) =>
  query
  | trusted(A) => send(inbox);
  end;
end.

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

We first 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. The second handler is the opposite: if the address is not trusted then its message is sent to the spam box of the recipient.

3 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 detect potential conflicts *statically* where the user supplies a *conflict declaration* such as

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

which explains to the system that the set of actions resulting from an event must not contain the actions `send(A)` and `send(B)` such that `A` and `B` are not equal. 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 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 [13], which is a platform that uses multiple external solver. 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. The analysis is formally proven to be sound for a core subset of the language, in the sense that the conflict detection scheme produces 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.

3.1 Syntax

We detail here only a core subset of SUPPL related to event handler bodies, as this is the only part that matters in our conflict detection scheme. As explained earlier, a handler body may contain commands to insert or delete rows from a table. However, as they are not involved in the conflict detection, these commands are not included below, 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 $a = x$".

$b :=$		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` $act_1(a_1, \ldots, a_n)$`,` $act_2(b_1, \ldots, b_m)$ `=>` qcl`.`	with a clause
	`conflict` $act_1(a_1, \ldots, a_n)$`,` $act_2(b_1, \ldots, b_m)$`.`	without a clause

3.2 Semantics

We now give semantics for an event handler's execution. Our goal is to describe when multiple actions can occur.

1. In a query, 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. In the case of a `foreach`, 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. 1 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`.

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

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

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

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

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

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

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

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

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

Fig. 1. Inference rules for $b \xrightarrow{\sigma} \ell$

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

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

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

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

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

Fig. 2. Inference rules for $\sigma \xrightarrow[qcl]{} \sigma'$

$\sigma \xrightarrow[qcl]{} \sigma'$ is defined in Fig. 2 and is a nondeterministic relation that means 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. 1 is an approximation of the actual execution of an event handler in the sense that for all actions a that are executed in b under the environment σ, there exists a ℓ such that $b \xrightarrow{\sigma} \ell$ and $a \in \ell$. However, it is not necessary true that there exists a ℓ such that for all actions a that are executed in b under the environment σ, $b \xrightarrow{\sigma} \ell$ and $a \in \ell$. This approximation is still suitable for the usage of our proof (in 3.5) as a property over all actions a that are executed in b under the environment σ can be proven by proving the property over all ℓ such that $b \xrightarrow{\sigma} \ell$.

3.3 Definition of conflicts

We define an *actual* conflict $(\sigma_1, a_1(x_1, \ldots, x_n)) \bowtie_c (\sigma_2, a_2(y_1, \ldots, y_m))$ as meaning that 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* where all z_k are substituted by $\sigma_1(x_k)$ and all t_k substituted by $\sigma_2(y_k)$ is true.

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 keep the rate of false positives minimal, we introduce a data structure named PConflict to represent *potential* conflicts.

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 defined by the 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. A control flow path is represented by a list of query clauses (as defined in 3.1). 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. The two paths may have a common prefix, which we store in *cp*. Consequently, 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 3.6.

A PConflict $(ev(z_1, \ldots, z_k), cp, d_1, d_2, a_1(x_1, \ldots, x_n), a_2(y_1, \ldots, y_m), \ell)$ corresponding to an actual conflict $(\sigma_1, a_1(x_1, \ldots, x_n)) \bowtie_c (\sigma_2, a_2(y_1, \ldots, y_m))$ is said to be *realizable* if there exists an environment σ such that the path *cp* is satisfied under the environment σ (noted $\sigma \models cp$), $\sigma_1 \models d_1$, $\sigma_2 \models d_2$, $\sigma \triangleright \sigma_1$, $\sigma \triangleright \sigma_2$ and $c \in \ell$. We will explain in detail in 3.6 what it means for a path to be satisfied by σ.

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

Fig. 3. Syntax of paths

3.4 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 actions whether they conflict. This is what the *is_ pconflict* function is used for in the definition of *combine*; 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 :

   ```
   pconflicts : Path → Body → PConflict List
   paths :      Body → (Actions, Args, Path) List
   combine :    Path → (Actions, Args, Path) List →
                (Actions, Args, Path) List → PConflict List.
   ```

   Here, *Path* is a control flow path as defined in 3.3 and Fig. 3.
   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.

   ```
   pconflicts pre b =
       case b of
       b_1;b_2 →                           pconflicts pre b_1 ++
                                           pconflicts pre b_2 ++
                                           combine pre (paths b_1) (paths b_2)
       foreach p(a_1,...,a_n) => b' →      pconflicts (p(a_1,...,a_n):pre) b' ++
                                           combine pre
                                               (paths (foreach p(a_1,...,a_n) => b'))
                                               (paths (foreach p(a_1,...,a_n) => b'))
       query (branch qcl b):bs →           pconflicts (qcl:pre) b ++
                                           pconflicts (¬qcl:pre) (query bs)
       query (defaultbranch b):bs →        pconflicts pre b
       _ →                                 []
   ```

   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 $q(a_1,\ldots,a_n)$ => $b$ →    $map\ (\lambda(act,\ args,\ p) \mapsto (act,\ args,\ q(a_1,\ldots,a_n){:}p))\ (paths\ b)$
    query (branch $qcl\ b$):$qbrs$ →     $map\ (\lambda(act,\ args,\ p)\mapsto (act,\ args,\ qcl{:}p))\ (paths\ b)$ ++
                                  $map\ (\lambda(act,\ args,\ p)\mapsto (act,\ args,\ \neg qcl{:}p))\ (paths$ (query $qbrs$))
    query (defaultbranch $b$):$qbrs$ → $paths\ b$
    query [] →                     []
    $act(a_1,\ldots,a_n)$ →               [($act$, $(a_1,\ldots,a_n)$, [])]
    _ →                            []
```

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.

```
$combine\ pre\ \ell\ \ell' = concat$
                $map\ (\lambda(act,\ args,\ p)\mapsto$
                  $(concat$
                    $map\ (\lambda(act',\ args',\ p') \mapsto$
                      case $is\_pconflict(act,\ act')$ of
                        [] → []
                        $c$ → [($pre$, $p$, $p'$, $act$, $args$, $act'$, $args'$, $c$)]) $\ell'$)) $\ell$
```

3.5 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* (as defined in 3.3) PConflict data structure. Let σ_0 be the empty environment, the theorem is stated as follows:

Soundness theorem:

$\forall b, w, \sigma_1, a_1, arg_1, \sigma_2, a_2, arg_2, c,$
$\quad b \xrightarrow{\sigma_0} w \wedge$
$\quad (\sigma_1, a_1(arg_1)) \in w \wedge$
$\quad (\sigma_2, a_2(arg_2)) \in w \wedge$
$\quad (\sigma_1, a_1(arg_1)) \bowtie_c (\sigma_2, a_2(arg_2)) \implies$
$\quad \exists cp, d_1, d_2, \ell,$
$\quad\quad (cp, d_1, d_2, a_1(arg_1), a_2(arg_2), \ell) \in pconflicts$ [] $b \ \wedge$
$\quad\quad c \in \ell \ \wedge$
$\quad\quad \exists \sigma, \sigma \models cp \ \wedge$
$\quad\quad\quad \sigma \triangleright \sigma_1 \ \wedge$
$\quad\quad\quad \sigma \triangleright \sigma_2 \ \wedge$
$\quad\quad\quad \sigma_1 \models d_1 \ \wedge$
$\quad\quad\quad \sigma_2 \models d_2$

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:

$\forall b, \sigma, w, b \xrightarrow{\sigma} w \rightarrow \forall(\sigma', a) \in w, \exists p, \sigma' \models p$ 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 $w = \ell$++ℓ' and (`foreach` $qcl\ b'$) $\xrightarrow{\sigma} \ell$ and $\sigma \underset{qcl}{\longrightarrow} \sigma'$ 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. Furthermore, the lemma and the theorem have both been verified in Coq [10].

3.6 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, table declarations are translated into the background theory; strings are the only tricky part as Why3 does not natively support equality over them; thus, strings 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`

`predicate trusted(address)`
`trusted("bob@leponge.com")`
	`predicate trusted (_arg0 : (address))`
	`   = (_arg0 = string4)`

`bob@leponge.com`	`constant string4 : string =`
	`  (Cons 98 (Cons 111 (Cons 98 ...`

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. However, had we not have a common path, but only divergent paths, the generated formula would look like :

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

This is what the previous analysis did. In this case, the solvers cannot find the contradiction, thus we get a false positive.

4 Feedback to the user

The conflict detection algorithm generates *potential* conflicts that are passed off to external SMT solvers, which verifies if the corresponding formulae are unsatisfiable and these conflicts 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 is happening. For example, suppose we have the two following paths from the policy described in section 2 highlighted in red and blue leading to a conflict that could not be ruled out.

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

```
handle receive(?A) =>
  query
  | isSpam(A) => send(spam);
  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 a 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`. `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, untrusted(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, so it is reported to the user by highlighting the rules and the control flow paths involved, as seen in Fig. 5. 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.

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).        predicate q(number, number).       predicate r(number, number).
p(N) :- N > 4.              q(N, M) :- N > M + 5.              r(N, M) :- M > 10.
p(N) :- N < 0.              q(N, M) :- M = 0.                  r(N, M) :- N > M + 2.

handle ev(?N, ?M) =>                          handle ev(?N, ?M) =>
  query =>                                      query =>
  | p(N) => query                               | r(N, M) => act2;
            | q(N, M) => act1;                  end;
            end;                              end.
  end;
end.
```

Fig. 4. One possible version of the potential conflicts

In the situation of Fig. 4, both versions of the conflict with combination of rules

```
p(N) :- N > 4.              p(N) :- N < 0.
q(N, M) :- N > M + 5.       q(N, M) :- N > M + 5.
r(N, M) :- N > M + 2.       r(N, M) :- N > M + 2.
```

cannot be ruled out. Therefore, both of the rules of predicate `p` are not 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 =>                                        query =>
  | friend("Gary", "Patrick") => act1;            | _ => act2;
end;                                            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 at the beginning of this section, we can add an axiom to tell the compiler that the conflict is not possible : `axiom trusted(A) -> (findall(U, untrusted(A, ?U), RS), set_size(RS) = 0)`. This says that if address `A` is trusted, then no user have filed complaint a 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 as solvers not managing to find an inconsistency does not mean there isn't one.

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, suppose that we have the following table declaration `table t(string, string) key(in, out)`, we would then generate the following `axiom t(A1, A2) -> t(B1, B2) -> A1 = B1 -> A2 = B2`. 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.

5 Related work

Conflict detection has been extensively studied, for example, Lupu and Sloman [14] 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 [12] to verify that there is no conflict within a security policy for firewalls. 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.

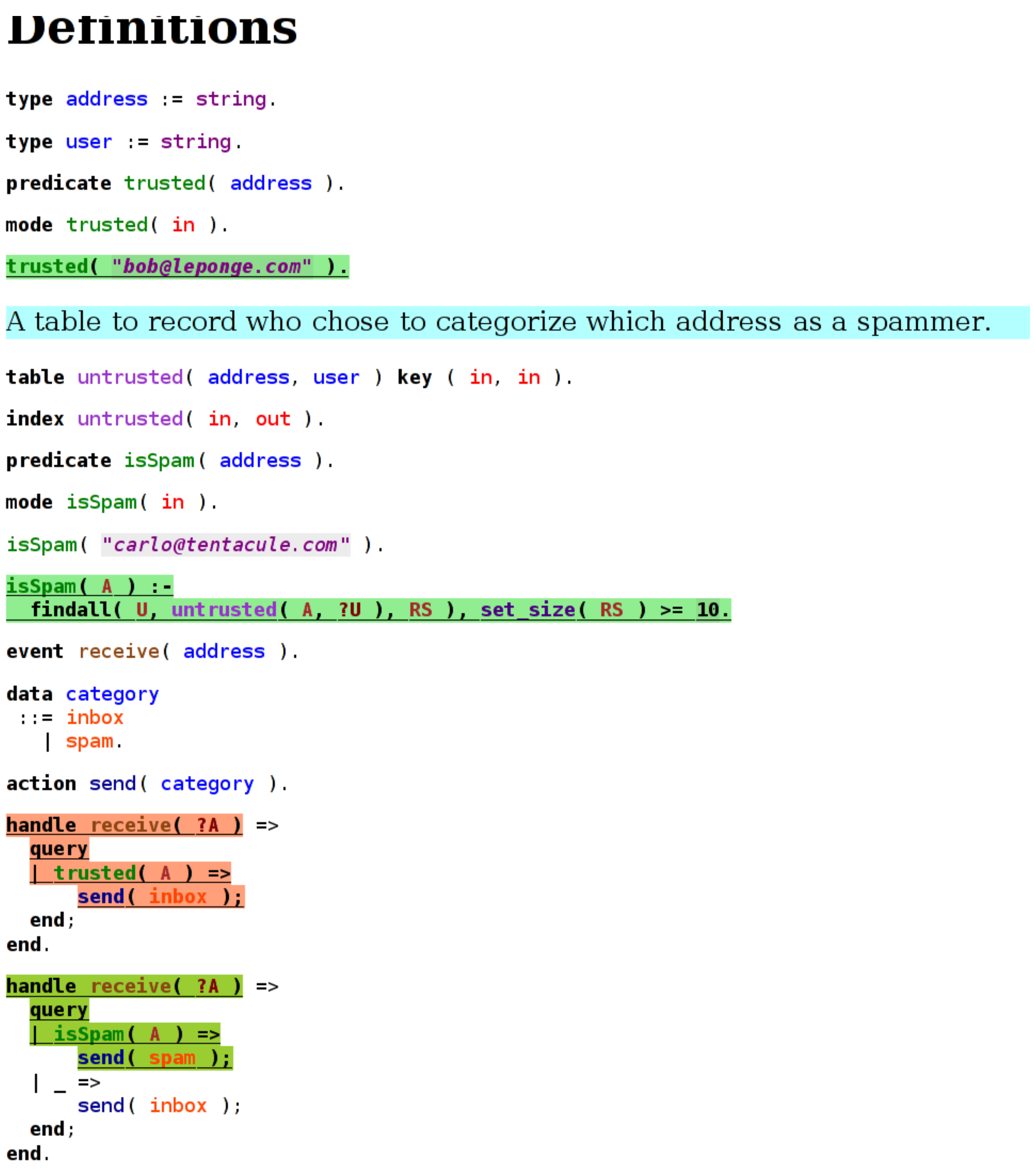
6 Conclusion

Suppl is an attempt to create a domain-neutral language for the ECA paradigm, while keeping the possibility of having static analyses. My internship took place in this context, an existing compiler written in Haskell which I improved by adding the diverse new features. I implemented and proved the soundness of the static control flow analysis algorithm for detecting conflicts in policies presented in section 3. I also designed and implemented a scheme to try helping users understand the causes of a conflict as well as improved the output to Why3 in order to decrease the number of false positives previously reported as presented in section 4.

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. Alessandro Armando, Jacopo Mantovani, and Lorenzo 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. Thomas Ball and Sriram K Rajamani. The SLAM project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.
3. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.
4. N Ben Youssef, Adel Bouhoula, and Florent Jacquemard. Automatic verification of conformance of firewall configurations to security policies. In *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*, pages 526–531. IEEE, 2009.



Definitions

```
type address := string.

type user := string.

predicate trusted( address ).

mode trusted( in ).

trusted( "bob@leponge.com" ).
```

A table to record who chose to categorize which address as a spammer.

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

index untrusted( in, out ).

predicate isSpam( address ).

mode isSpam( in ).

isSpam( "carlo@tentacule.com" ).

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

event receive( address ).

data category
 ::= inbox
   | spam.

action send( category ).

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

handle receive( ?A ) =>
  query
  | isSpam( A ) =>
      send( spam );
  | _ =>
      send( inbox );
  end;
end.
```

Fig. 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 green in the handlers.

5. Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.
6. François Bobot, Sylvain Conchon, Evelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover. `http://alt-ergo.lri.fr`, 2008.
7. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.
8. Alain Colmerauer. An introduction to prolog III. In John W. Lloyd, editor, *Computational Logic*, ESPRIT Basic Research Series, pages 37–79. Springer Berlin Heidelberg, 1990.
9. Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer Berlin Heidelberg, 2005.
10. The Coq development team. Coq. `http://coq.inria.fr/`.
11. Robert Dockins and Andrew Tolmach. SUPPL: A flexible language for policies. In *Proc. of the 12th Asian Symposium on Programming Languages and Systems (APLAS 2014)*. Springer, 2014 (To appear).
12. Bruno Dutertre and Leonardo de Moura. Yices. `http://yices.csl.sri.com/`.
13. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.
14. Emil C Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *Software Engineering, IEEE Transactions on*, 25(6):852–869, 1999.