Suppl : A Flexible Language for Policies

Robert Dockins and Andrew Tolmach

Portland State University

Abstract. We present the Simple Unified Policy Programming Language (Suppl), a domain-neutral language for stating, executing, and analyzing event-condition-action policies. Suppl uses a novel combination of pure logic programming and disciplined imperative programming features to make it easy for non-expert users to express common policy idioms. The language is strongly typed and moded to allow static detection of common programming errors, and it supports a novel logic-based static analysis that can detect internally inconsistent policies. Suppl has been implemented as a compiler to Prolog and used to build several network security applications in a Java framework.

1 Introduction

Many computing systems incorporate *policies* that specify how the system should respond to events. Policies are used to define, e.g., who may access protected web sites, how to categorize arriving emails, or what to do when the temperature in boiler #2 exceeds safe limits. Because policies change over time, designers often provide a mechanism to express them separately from the main body of implementation code. This mechanism might be simple, like configuration parameters accessed by a GUI (e.g., your email client); but it may be a non-trivial external language in its own right (e.g., configuration files for a Cisco router). A dedicated policy language allows relatively non-technical users to write and review policies without understanding the underlying code. It may also support automatic analysis of policies for properties such as consistency or completeness.

Many existing policy languages evolved in the context of particular applications or execution environments and hence are domain-specific, "baking in" concepts related to, say, networks or access control. However, policy languages often share common basic requirements and structures. This raises a natural challenge: can we define a *domain-neutral* policy language suitable for use in a wide variety of applications? Moreover, existing policy languages often appear very ad-hoc: they typically lack control abstractions, types, and support for modularity. This raises another challenge: can we improve on these languages by applying ideas from *programming language design*?

Suppl, the Simple Unified Policy Programming Language, is our attempt to address these challenges.[1] Suppl is designed to describe the large class of policies known as *event-condition-action (ECA)* policies. The ECA paradigm, originally

[1] http://web.cecs.pdx.edu/~rdockins/suppl/

developed in the context of active databases [10], is based on an event-handling loop. When an external stimulus generates an event, the policy evaluates conditions based on the current state of the world and its internal memory and decides what actions to take. SUPPL uses a novel combination of (pure) predicates from logic programming, used to describe conditions, and imperative event handlers, which generate actions. Both parts work together to make expressing common policy idioms simple and understandable. The SUPPL language is parameterized over the vocabulary of events and actions needed for a particular domain. These are provided by an ambient execution environment (coded in a conventional language) which triggers calls into SUPPL when an event occurs and interprets the action directives that SUPPL returns.

SUPPL is strongly typed, strongly moded, and locally stateless. These features are designed to make SUPPL programs easy to reason about and to facilitate the early detection of errors. Despite its locally stateless properties, SUPPL is capable of expressing stateful policies by making controlled use of *data tables* that provide a principled point of interaction between the stateless logic-programming core and the imperative event-handling language.

SUPPL is also designed to allow easy combination of distinct policy units, perhaps written by different people. Both predicates and event handlers can be easily extended by additional, textually separate, clauses. However, these features make it possible to write policies that are incoherent—for example, an access control policy might generate both "allow" and "deny" actions in response to a request event. To report such possible inconsistencies, we have developed a novel logic-based static analysis called *conflict detection*, which is only feasible because we have a carefully-designed language specifically for policies.

SUPPL has been implemented as a compiler generating Prolog code, which runs in a Java execution environment that provides the realizations of events and actions. On top of this implementation we have built two network security applications. The first is a prototype active network firewall built on the Linux netfilter stack, in which connection attempts are mediated by a SUPPL policy. The second is the SOUND platform [9], which uses active sensing to detect misbehavior on networks and introduction-based-routing [13] to control access. SUPPL can be used to define various aspects of policy in this system, for example, what remedial actions to take when misbehavior is detected.

The detailed contributions of this paper are as follows:

– A tutorial introduction to SUPPL from the viewpoint of a policy author, using a simple example (§2).
– A novel approach to integrating pure predicates and stateful event handlers (§3.1).
– The static type and mode system used for predicates, which is both simple and practical (§3.2).
– Conflict detection analysis, which combines control-flow analysis and automated provers to find potential inconsistencies in policies (§4).
– An implementation of SUPPL, using a Java-based runtime system (§5).

2 Suppl by example

Suppl is our attempt to build a general-purpose policy language as described in the introduction. It explicitly embraces the ECA paradigm; events and actions are primitive concepts in the language, and event handlers are the fundamental construct for initiating computation. Conditions are another bedrock concept: the main programming abstraction in Suppl is the predicate, similar to that found in logic programming languages like Prolog. Unlike Prolog, the Suppl predicate language is pure (no side effects), strongly typed and strongly moded. Event handlers are written in a separate imperative vocabulary designed to make expressing policy decisions as natural as possible.

To illustrate Suppl, we will examine an extended example. Suppose we are writing a policy for a system that controls door locks in a secure facility. A person requests a door to open by using their keycard; the system decides to accept the request and open the door, or to deny the request and leave the door locked. The system is also capable of raising an alarm, which will cause security personnel to head to the area to investigate.

Primitives We can model these concepts in Suppl in a few lines; see Listing 1, lines 1–8. We declare `person`, `scanner`, `location` and `door` to be primitive types. These types will have some concrete implementation in the security system, but they are treated as opaque by Suppl. We also declare an event `open_door_request`, indicating that someone has used a keycard scanner and requested a door to be opened, and two actions the system can take in response: `open_door` and `dispatch_security`. These declarations (together with the other primitive declarations) form the interface between the policy and the system being governed. Note that a policy may decide to do *nothing* in response to an event; for this door lock setting, this constitutes a request denial.

On line 10 we declare that the `open_door` and `dispatch_security` actions are in *conflict*. This is our way to state our intention that a single policy event should not elicit both actions. Conflict declarations will come into play when we discuss conflict analysis later.

To define any interesting policies regarding this security system, we need to have some operations that allow us to examine the properties of the opaque types. For example, we need to know which scanners govern which doors, where the scanners are, and the location to which the scanner gates access. Lines 12–14 of Listing 1 declare three functions from the opaque type `scanner` to `doors` and `locations`. It will eventually be the responsibility of the security system to implement these operations. Finally, we need to know who is allowed to be where. Line 16 introduces a *predicate*, `authorized_loc`, which represents a relation between persons and locations. For now, we leave unspecified how authorizations are determined; thus the predicate is declared `primitive`. The `in` keyword is related to the mode system and indicates that uses of this predicate must pass both arguments in; modes are discussed in more detail below.

```
 1 primitive type person.
 2 primitive type scanner.
 3 primitive type location.
 4 primitive type door.
 5
 6 event open_door_request(person, scanner).
 7 action open_door(door).
 8 action dispatch_security(location).
 9
10 conflict open_door(_), dispatch_security(_).
11
12 primitive function scan_door(scanner) yields door.
13 primitive function scan_loc(scanner) yields location.
14 primitive function scan_gates(scanner) yields location.
15
16 primitive predicate authorized_loc(person in, location in).
17
18 handle open_door_request(?P, ?S) =>
19  query
20  | authorized_loc(P, scan_loc(S)) =>
21    query
22    | authorized_loc(P,scan_gates(S)) => open_door(scan_door(S));
23    | _ => skip;
24    end;
25  | _ => dispatch_security(scan_loc(S));
26  end;
27 end.
```

Listing 1. A simple door security policy

Event Handler Now we can define a simple event handler for the security system (lines 18–27). This handler says how to respond to an `open_door_request` event. Like every event handler, it starts by naming the event to be handled and binding the event arguments; the `?P` form indicates a variable binding. The main body of the handler consists of a query statement with two branches. Each branch consists of a logical query on the left of the `=>` symbol and a list of statements on the right. The first branch is entered if the person `P` is authorized to be where they are now, i.e., in the location where the scanner is; otherwise the second branch is entered and security is dispatched to that location. In the first branch, another query is run to see if person `P` is allowed on the far side of the door. If so, the door is opened; otherwise, the request is denied. In general, a query construct may have many branches; the queries are attempted in order and (only) the first one to succeed is executed. The underscore represents a trivial query that always succeeds, and `skip` is a command that has no effect. If no branch of a query construct succeeds, nothing happens. Thus, the query branch on line 23 is redundant and could be eliminated without changing the meaning of the program.

Authorization Suppose we want to define the predicate `authorized_loc` instead of making it a primitive. To do this, we remove the `primitive` keyword from its declaration and we specify rules that define when the predicate holds.

The syntax for rules is quite similar to Prolog syntax. In particular, we adopt the Prolog lexical convention that variables begin with uppercase letters and program identifiers begin with lowercase letters.

Listing 2, lines 1–9, uses two rules to define the `authorized_loc` predicate in terms of some new, lower-level, primitives. (Note: for space reasons we have not repeated lines 1–14 of Listing 1.) A rule consists of a single predicate applied to some arguments followed by the `:-` symbol and a comma-separated list of clauses. A rule should be read as an implication from right to left. Thus, the rule `authorized_loc(P,L):- public_space(L)` means that "for all `P` and `L`, if `L` is a public space then `P` is authorized to be in `L`." When multiple clauses are separated by a comma, all must hold. So the second rule means that `P` is authorized to be in `L` if `P` belongs to some group `G` that owns `L`. Finally, the meaning of the predicate `authorized_loc` is the disjunction of all the right-hand-sides of its rules. So, `authorized_loc` holds if either of its two rule bodies hold.

The overall effect of this policy will be to allow persons into and out of areas that are public or for which they are members of an owning group. If someone gets into an area for which they are not authorized (by tailgating someone else, say) then security will be notified if they try to leave by using a keycard scanner.

Detecting repeated failures Now, suppose we want to prevent someone from doing a trial-and-error scan with their keycard; that is, we don't want people to be able to map out which doors are opened by a keycard by simply trying all of them and seeing which ones open. Such a pattern of use might occur if a keycard is stolen and the thief doesn't know what doors it opens. One way to mitigate this risk is to keep track of failed open attempts. If too many failed attempts happen within a short time frame, we want to dispatch security to investigate.

To do this, we need to keep some state about failed requests. SUPPL is, by design, locally stateless, so there are no mutable references or data structures we can manipulate within queries to keep track of this information. Instead, SUPPL includes a concept of *data tables*, which provide a principled way to implement stateful policies. From the point of view of the logic programming query language, tables are just another predicate that may be used in rules. However, the imperative event handling language has commands that insert and delete rows from tables.

Suppose we want to trigger an alarm if more than five failed attempts are made by a single person within an hour. To keep track of the required data, we set up a table and write an event handler to populate it (see Listing 2 lines 11–21). Table declarations are similar in most ways to predicate declarations; the columns of the table are given as an ordered tuple of types, just as for predicates. However, unlike predicates, tables behave much like the tables of a relational database: tuples are added and removed from tables explicitly rather than by defining rules. The `key` clause declares the table's primary key. The mode keywords following `key` indicate which columns form the table's primary key: columns declared with the mode `in` are in the primary key and those declared with mode `out` are not. Every table will contain at most one row for

```
predicate authorized_loc(person in, location in).

authorized_loc(P,L) :- public_space(L).
authorized_loc(P,L) :- group_owns(G,L), group_member(P,G).

primitive type group.
primitive predicate public_space(location in).
primitive predicate group_owns(group in, location in).
primitive predicate group_member(person in, group out).

table failed_attempts(person, scanner, eventid)
   key (in,in,in) lifetime 3600000.
index failed_attempts(in, out, out).

handle open_door_request(?P, ?S) =>
  query
  | authorized_loc(P, scan_gates(S)) => skip;
  | _ => queue insert (P, S, current_event)
           into failed_attempts;
  end;
end.

predicate excessive_failures(person in).
excessive_failures(P) :-
 findall(EID,failed_attempts(P,_,?EID),RS), set_size(RS) >= 5.

handle open_door_request(?P, ?S) =>
 query
 | authorized_loc(P, scan_loc(S)) =>
   query
   | authorized_loc(P, scan_gates(S)) =>
         open_door(scan_door(S));
   | excessive_failures(P) =>
         dispatch_security(scan_loc(S));
   end;
 | _ => dispatch_security(scan_loc(S));
 end;
end.
```

Listing 2. A more complicated door security policy

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. Tables also have an optional `lifetime` argument that indicates how many milliseconds each row should remain in the table from the time it was inserted (3600000 milliseconds corresponds to one hour). The `eventid` type is a built-in type that is used to give a unique identifier to each event occurrence.

The `index` declaration indicates that we intend to query this table by supplying the first column as an argument; the index declaration both interacts with the mode system (described below) and also suggests to the implementation that building an index for this table on its first column would be worthwhile. Unlike the primary key, a table index does not impose any uniqueness constraints.

Despite the strong similarities between SUPPL tables and the relational tables of a typical RDBMS, their use cases are rather different. SUPPL tables are primarily intended to store short-term, "soft" data; the SUPPL runtime holds table data in memory and makes no persistence guarantees about it. Restarting the SUPPL runtime will clear all table data. It should be possible to have SUPPL data tables backed instead by a persistent RDBMS; however, a reasonable semantics for interacting with external RDBMSs seems to require distributed transaction support in the general case. We hope to examine these issues in future work.

Now we write an event handler that inserts a row into `failed_attempts` whenever an unauthorized person attempts to enter a gated area (Lines 15–21). It is normal in SUPPL to have more than one handler for a given event; when that event occurs, *all* its handlers will be run. The query illustrates the use of sequential evaluation to implement a form of negation. If the person is authorized, the first query branch succeeds and the handler does nothing; otherwise, the second branch is executed and the insertion is performed. The primitive `current_event` function returns the `eventid` corresponding to the event currently being handled. The result of this pattern is that we get a sliding window view of all the failed open attempts that have occurred in the last hour.

Note that the command to insert a row is written `queue insert`: this indicates that the insert does not happen immediately. Instead, it occurs after all handlers for the current event have completed. This is to ensure that there are no complicated and difficult-to-debug interactions between separately-defined event handlers. State changes are queued up and executed after all handers are finished, so that the next event that occurs will see the updated table state.

Now we can write the `excessive_failures` predicate that holds if a person has amassed too many failed attempts (lines 23–25). This predicate holds on a person `P` who has five or more distinct failed door-open event identifiers in the `failed_attempts` table. The `excessive_failures` predicate relies on the primitive `findall` construct, which calculates a set of all the solutions to a given query. Here we use it to get a result set whose size we can then calculate using the built-in `set_size` function. As used here, the `findall` can be rendered as "find all instances of `EID` such that `P` is related to `EID` (for some ignored scanner value) in the failed events table; place the result set in variable `RS`." In contrast to every other predicate construct, `findall` has explicit variable binding. Variables bound in the second argument (the search goal) may appear in the first argument. Using this predicate, we can now replace our original event handler (Listing 1 lines 17–27) with one that also responds to `excessive_failures` (Listing 2 lines 27–38).

3 Suppl in Detail

SUPPL's design attempts to balance competing objectives: simplicity, expressivity, support for early detection of errors, and ease of combining separately-written policies. The use of logic programming, for example, is driven both by the need for expressivity (realistic policy conditions are naturally expressed using logic programming rules) and to make it easy to combine policies. As compared to procedures or functions, it is easy to extend the functionality of predicates by adding new rules. In the interests of both simplicity and expressivity, we allow arbitrary recursive predicates to be written, which makes the language Turing-complete. Event handling is likewise easy to extend by adding new handlers—event handling logic does not have to be collected together in a single place.

A slightly simplified syntax for SUPPL is presented in Figure 1. For lack of space, we do not give full explanations of all the language's constructs, but instead focus on the most important and novel. There are four major syntactic classes: terms, clauses, handler bodies and declarations. Terms represent data values, clauses are used to define predicates, and handler bodies are used to implement handlers and procedures. A SUPPL program consists of a set of declarations, which are used both to provide static information to the compiler (declaring types and modes for predicates, functions, etc.) and to implement the policy (rules, event handlers, procedure definitions). Terms are quite similar to those of Prolog, with the addition of the variable binding form `?A` (used inside handler bodies to make variable bindings explicit), and of tuple data structures. Clauses also take inspiration from Prolog; the main syntactic difference is that disjunction is written with a vertical bar rather than with the traditional semicolon. The operational semantics of the logic programming core of SUPPL can be understood in a standard way, as performing selective linear definite clause (SLD) resolution [18] with negation-as-failure [7]. The parts of SUPPL that cannot be understood by analogy to standard logic programming concepts are covered in further detail below.

3.1 Event handlers

The primary interface between a SUPPL policy and the system it governs is defined by *events* and *actions*. These are declared as distinguished identifiers carrying some number of data arguments. Their meaning is determined entirely by the surrounding execution environment.

Program execution is always initiated by an event and events happen when the system being governed wishes to interrogate the policy. When an event occurs, every event handler in the program matching the event is executed and the set (possibly empty) of all resulting actions is collected together to be passed to the surrounding execution environment. The execution environment is responsible for executing these actions, as well as for implementing all declared primitive functions and predicates. The SUPPL semantics assumes that the execution of primitive functions and predicates is side-effect free. SUPPL is "locally stateless," which means the only state in SUPPL is in the data tables, and they do not change during the execution of the handlers for a single event. Instead, the effects of any `queue insert` or `queue delete` statements are delayed until after all handlers for the event have completed.

```
d ::=                                                                              Declaration
   | primitive type ⟨id⟩.                                                      prim type decl
   | type ⟨id⟩ := t.                                                                type decl
   | data ⟨id⟩ ::= ⟨id⟩(t1, ··· , tm) | ··· | ⟨id⟩(t1, ··· , tn).              data type decl
   | event ⟨id⟩(t1, ··· , tn).                                                     event decl
   | action ⟨id⟩(t1, ··· , tn).                                                   action decl
   | conflict ⟨id1⟩(t1, ··· , tn),⟨id2⟩(t1, ··· , tm) (=> c)? .                 conflict decl
   | procedure ⟨id⟩(t1, ··· , tn).                                             procedure decl
   | primitive function⟨id⟩ (t1, ··· , tn) yields t .                     prim function decl
   | (primitive)? predicate ⟨id⟩(t1 o1?, ··· , tn on?).                        predicate decl
   | mode ⟨id⟩(o1, ··· , on).                                                       mode decl
   | table ⟨id⟩(t1, ··· , tn) key (o1, ··· , on) (lifetime ⟨int⟩)?.                table decl
   | index ⟨id⟩(o1, ··· , on).                                                     index decl
   | g :- c .                                                                            rule
   | handle ⟨id⟩(?X1, ··· , ?Xn) =>  b  end.                                    event handler
   | define procedure ⟨id⟩(?X1, ··· , ?Xn) :=  b  end.                         procedure defn
   | axiom c .                                                                     axiom decl
   | lemma c .                                                                     lemma decl

b ::=                                                                             Handler Body
   | b1 b2                                                                           sequence
   | ⟨id⟩(m1, ··· , mn);                                                  procedure or action
   | queue insert(m1, ··· , mn)  into ⟨id⟩;                                      table insert
   | queue delete(m1, ··· , mn)  from ⟨id⟩;                                      table delete
   | skip;                                                                               noop
   | query | c1 => b1  ··· | cn => bn end;                                  multibranch query
   | foreach c => b end;                                                        foreach query

t ::=                             Type     o  ::=                                         Mode
   | ⟨id⟩                   named type        | in | out | ignore
   | X, Y, Z, ···       type variables
   | list(t)                      list     g  ::= ⟨id⟩(m1, ··· , mn)                      Goal
   | set(t)                 finite set
   | map(t1, t2)            finite map     m ::=                                          Term
   | t1 ∗ ··· ∗ tn          tuple type        | "literal", ···                     strings
   | number               numeric type        | 10, 3.14, 2.9e8, ···               numbers
   | string                string type        | X, Y, Z, ···                     variables
                                              | ?X, ?Y, ?Z, ···               var bindings
c ::=                           Clause        | _                            anonymous var
   | m1 = m2 | m1 <> m2  (dis)equality        | ⟨id⟩(m1, ··· , mn)           function call
   | m1 <= m2 | m1 < m2   comparisions        | m1 + m2 | m1 − m2              numeric ops
   | m1 >= m2 | m1 > m2                       | m1 ∗ m2 | m1/m2
   | ⟨id⟩(m1, ··· , mn)      predicate        | ~m                                negative
   | c1 | c2               disjunction        | [ ]                             empty list
   | c1, c2                conjunction        | [m1, ··· , mn]               concrete list
   | not c                    negation        | [m1 | m2]                   list cons cell
   | c1 -> c2              implication        | (m1, ··· , mn)                       tuple
   | findall(m, g, X)         find all
```

Fig. 1. Simplified syntax of Suppl

All program execution is event-driven, and the event handler serves as the entry point for Suppl programs. The body of an event handler is a sequence of statements, which may be actions, commands to manipulate data tables, `query` evaluations, or `foreach` invocations. Event handlers can also invoke user-defined procedures that abstract over common sequences of statements.

The `query` construct, illustrated by several examples in §2, consists of a series of branches, each guarded by a query into the core logic-programming part of the language; the branch corresponding to (just) the first successful query is executed. This behavior captures a common idiom that is inconvenient to express in pure logic programming (without cut).

The `foreach` construct `foreach some_pred(A,?B)=> ... end;` is an iterator: it asks the system to find all values for `B` such that `some_pred(A,B)` is true, and executes its body once for each instantiation found.

3.2 Predicates, Types, and Modes

Unlike Prolog, Suppl predicates are pure (they lack both side-effects and non-logical constructs, like cut), well-typed and well-moded. Types and modes are primarily intended to help with early detection of errors. They make large classes of "shallow" errors (e.g., mixing up argument order) detectable at compile time. A strong typing discipline also makes it easier to interface with SMT solvers for discovering deeper program properties (see §4). Our type and mode systems are similar to those of Mercury [22] and HAL [14], but significantly simpler.

Types built in to the system include `number` and `string`. There are also built-in polymorphic type operators `list`, (finite) `set` and (finite) `map`. Users may also declare recursive algebraic datatypes for generating arbitrary tree-shaped data structures. Every predicate in a Suppl program must be declared, giving the number and types of its arguments.

Modes indicate which arguments of a predicate are inputs and which are considered outputs. For example, the predicate call `member([1,2,3,4], 2)` asks the question: "does the list `[1,2,3,4]` contain the value 2?" Both arguments are used in input mode. On the other hand, the call `member([1,2,3,4],N)` asks the system to *find* all values for `N` (four in this case) that make the statement true. Here we are using the second argument in output mode. Not all modes make sense for a given predicate. The call `member(L, 5)` asks the system to find all lists `L` that contain value 5; there is no obvious algorithm for doing this, so `member` can not be used with its first argument in output mode.

As with types, the modes of all predicates in a Suppl program must be declared. For example, we can express the allowed modes for the `member` predicate by writing:

```
mode member(in, in).
mode member(in, out).
```

The rules of predicates are checked to ensure they respect the specified modes by reordering the body of each rule (if necessary) so that every variable is instantiated before it is used. Variables get instantiated by being passed in as formal arguments to a predicate rule, by being generated as outputs from predicate calls, or via the equality operator. Mode checking ensures that every predicate can be implemented as a nondeterministic program manipulating only ground data (i.e., containing no unbound variables) and ensures that "instantiation errors" (which can happen in an ill-moded Prolog program) never occur.

4 Conflict detection

Problem The extensibility of predicates and event handlers makes it easy to combine Suppl code from multiple sources, but also makes it easy to write policies that are self-contradictory. The runtime environment must choose *some* action (even if that is to do nothing) in response to an incoherent policy outcome; but without further guidelines, any such choice is necessarily arbitrary.

Consider again the door-lock policy from section 2. The main event handler (see Listing 2 lines 27–38) opens the door if the requester is authorized both to be where he is *and* where he is going. Security is instead dispatched if the user is not authorized to be where he is. Now suppose we separately want to define a special class of persons that always have access to any door. One way to do this is to add the following predicate and handler. We assume the environment has some way to determine who currently has global privileges.

```
primitive predicate has_global_privileges(person in).

handle open_door_request(?P, ?S) =>
  query
  | has_global_privileges(P) => open_door(scan_gates(S));
  end;
end.
```

Each of these handlers make sense on their own, but in combination they can result in the policy both opening a door (because the requester has global access) and also dispatching security (because the requester is not authorized according to `authorized_loc`). Such a result is undesirable.

Solutions One solution might be to layer an additional mechanism for dynamic conflict *resolution* on top of the basic policy language. For example, we might provide a way to assign priorities to actions, and say that higher-priority actions "win" in the event of a conflict. But the details of such an approach become complicated: it is hard to find a modular way to assign priorities (especially because ties must not be allowed), and it is not clear what to do about the actions that "lose." Dynamic conflict resolution can lead to fragile, inscrutable policies where minor-seeming changes have wide-ranging, poorly understood effects.

We would prefer instead to provide a tool that detects potential conflicts *statically*, so that the policy programmer can then use the existing facilities of the policy language to fix them before execution. Specifically, we focus on a static analysis that identifies control-flow paths through a policy that are initiated by the same event and lead to conflicting actions. The policy author declares what actions she considers conflicting by writing a conflict declaration, e.g., listing 1 line 10.

Let us examine the example conflict from above in more detail. For the conflict to occur there must be some event that triggers both handlers; thus, assume `open_door_request(P,S)` has occurred. The first handler must have control flow pass to one of the two branches that dispatches security. For now, let us consider only one of these, the one appearing in the outermost `query` construct. For this branch to activate, the previous branch must have failed, so `authorized_location(P,scanner_loc(S))` is false. However, the proposition `has_global_privileges(P)` must hold for the other handler to issue the conflicting `open_door` verdict. To rule out this conflict, we must prove a contradiction under these assumptions. However, we cannot do this; nothing in the definition of `authorized_location` allows us to derive a contradiction. So our analysis should report a possible conflict between the two handlers.

We have developed a prototype conflict detection analysis for SUPPL that formalizes the line of reasoning outlined above. The analysis works in two phases. In the first phase, it identifies all the pairs of control-flow paths in the program that could possibly conflict. For each of these, it builds a formula in first-order logic that states what conditions would have to be true for the program to traverse both paths on a single event occurrence. In the second phase, these formulae are passed to an off-the-shelf SMT solver; we have experimented with Z3 [21], CVC4 [1] and Alt-Ergo [2]. If the solver can show the formula is unsatisfiable, we know the potential conflict cannot occur. Otherwise (if the solver finds a model or runs out of time), we report a potential conflict to the user.

We have designed the analysis to be sound, in the sense that it reports all potential conflicts. But to be useful in practice, it is crucial that the analysis also be as precise as possible, so that false positives are rare. Because SUPPL is Turing-complete, the analysis cannot be complete, in the sense that it only reports genuine conflicts: some false positives are inevitable. Moreover, the particular SMT solvers we use may have limitations that induce further imprecisions. However, although we are still in the early stages of working with our prototype, our initial results on precision are promising.

Generation of Conflict Formulae The problem definitions that get fed to the external solver break down into two distinct parts. One part is the definition of predicates in the program, which we call the background theory. This theory is the same for all problem instances. The second part consists of a formula corresponding to a particular pair of potentially-conflicting control-flow paths.

Building the background theory follows well-known work in the semantics of logic programs with negation-as-failure. For each defined predicate, the analysis calculates the Clark completion [7], which is a standard way to render the semantics of a logic program into a formula of first-order logic. It essentially formalizes the idea that a predicate is defined by the disjunction of its rules, while taking care to bind variables in the places that give the desired meaning. In other words, the Clark completion defines a predicate to hold if and only if it is established by one of its rules. Primitive predicates are uninterpreted in the translation; that is, they are declared but not given any definition. The Clark completion procedure is sound (but not complete) with respect to Selective Linear Definite clause (SLD) resolution, the logical reasoning system underlying the operational semantics of Prolog and similar logic programming languages [18]. This means that every query answered by SLD resolution will be a model of the Clark completion. However, in some cases SLD resolution will fail to terminate even when the Clark completion has a model.

The soundness of Clark completion is sufficient for the soundness of our conflict analysis. Our analysis attempts to show that the Clark completion has *no* models corresponding to the control-flow paths in question; *a fortiori* a logic-programming language based on SLD resolution will fail to activate those control-flow paths. Consider, for example, the authorized_loc predicate, defined by the rules below.

```
authorized_loc(P,L) :- public_space(L).
authorized_loc(P,L) :- group_owns(G,L), group_member(P,G).
```

The Clark completion defines this predicate by the first-order formula below:

$$\forall P\ L.\ \texttt{authorized\_loc}(P,L) \leftrightarrow \\ \big(\texttt{public\_space}(L)\ \vee\ (\exists G.\ \texttt{group\_owns}(G,L) \wedge \texttt{group\_member}(P,G))\big)$$

Note that variables corresponding to the predicate arguments are quantified universally at the outside, whereas variables appearing only in the body are quantified existentially at the level of the rule. If a rule body contains a compound term instead of a variable, a new fresh variable is introduced and an equality is added to the rule body.

Next we examine the control-flow paths through the imperative event handlers so we can generate queries to send to an SMT solver. This is done via a recursive algorithm which, when given the syntax of a handler body, calculates a set of *potential conflicts*. A potential conflict consists of the following data: the name of the initiating event, the user-defined conflict clause that is involved, and control flow paths that lead from the initiating event to the conflicting actions. From a given control-flow path, we can determine what logical queries must have succeeded and failed for the control-flow path to be traversed. For example, if a control-flow path goes into a branch of a query construct, the logical predicates guarding that branch must hold; and furthermore, the logical predicates guarding any preceding branches in the query must fail.

For each potential conflict, we can construct a formula in first-order logic that represents the state of affairs that must exist for the potential conflict to actually occur. For the example above, the generated conflict formula is:

$$\exists P\ S. \\ \neg\texttt{authorized\_loc}(P,\texttt{scan\_loc}(S))\ \wedge\ \texttt{has\_global\_privileges}(P)$$

A potential conflict is *satisfiable* if the associated conflict formula is satisfiable, given the background theory of the associated logical predicates. Dually, a potential conflict is *unsatisfiable* if we can derive a contradiction by assuming the conflict formula; in other words, if it is logically impossible for the potential conflict to actually occur.

We have proved the soundness of our conflict analysis with respect to an idealized version of the semantics of SUPPL. In particular, we have proved that, for every actual conflict that occurs during the run of a SUPPL program, our analysis algorithm generates a satisfiable potential conflict. A straightforward corollary is: if all the potential conflicts generated by the conflict analysis are unsatisfiable, then the policy will produce no actual conflicts when executed. We lack here the space to discuss the conflict generation algorithm or its proof; details will appear in a forthcoming paper [23].

Asserting facts Sometimes the conflict detection system will report a conflict where none exists because it has no way to analyze the policy primitives. Policy authors can communicate domain knowledge about the primitives to the analysis by using the `axiom` keyword. Any clause asserted as an axiom is assumed to be true and will be used by external provers during analysis. Of course, the user must be very careful only to assert axioms that actually hold; otherwise the correctness of the analysis will be compromised.

A policy author can also state a `lemma`; like axioms, lemmas are used by provers when trying to discharge proof obligations. However, the prover will also try to prove the lemma. In this way, the policy author can help guide provers toward finding useful facts they might not otherwise find in time, and also document the policy with properties that are expected to hold.

External Solver To interface with back-end provers, we use the Why3 program verification system [4]. Why3 understands all the concepts we need to express SUPPL programs: first-order logic, recursive datatypes, parametric polymorphism, numbers, sets, etc. Why3 can translate all these concepts into forms that can be understood by back-end SMT solvers; in particular, Why3 knows how to perform the tricky transformations that are needed to remove parametric polymorphism, which is not supported natively by most SMT solvers (Alt-Ergo seems to be the sole exception [3]).

Once our conflict detection problems are exported in Why3 format, we can use the Why3 system to dispatch the problems to a variety of solvers, including: CVC4, Alt-Ergo, Z3, and many others. Problems may even be translated into a form suitable to manual proof in Coq or Isabelle/HOL, if desired.

Discussion We cannot hope to have a complete procedure for finding conflicts, and false positives are inevitable. However, even if the problem were decidable, using SMT solvers means that, as a practical matter, we cannot expect to always get back answers in a reasonable amount of time. Nonetheless, our limited experience so far has given us promising results; CVC4 and Alt-Ergo both seem to do well at discharging the problem instances we build. We tested a number of different ways to resolve the conflict in our door lock policy from above (and for other similar policies); for each alternative we tried, a solver was able to prove the conflict could not occur using less than 1 second of runtime.

We do not yet have any data about how this analysis system scales to large policies. The number of potential conflicts is quadratic in the number of control-flow paths in a program, but this may be acceptable for realistic policies.

Conflict detection for policies is important in its own right. However, the potential applications for our analysis pipeline go further. For example, lemmas can be used simply to document properties of a policy that a user expects to be true; over time, as a policy is modified, if the lemma is falsified by some change, the analysis will indicate if the lemma can no longer be proved, indicating a problem. In future work we hope to explore other avenues for analysis, including liveness properties and data invariants.

5 Implementation

The implementation of Suppl is divided into two parts: a compiler that translates Suppl code into an executable Prolog policy; and a backend runtime system. The compiler is a standalone application written in Haskell, whereas the runtime is built on top of the tuProlog interpreter [11], which is written in Java. Suppl is an open-source project; additional information may be found at the first author's home page.[2]

The most complicated tasks performed by the frontend compiler involve implementing the static type and mode disciplines. The type system is essentially a first-order variant of Hindley-Milner polymorphism. The type checking algorithm follows the main ideas of the classic type inference algorithm W [20].

The mode system is responsible for ensuring that each predicate defined in a policy respects its stated modes. Actually, the term "mode checking" is a slight misnomer, because each mode for a predicate causes different code to be generated. Mode checking works by literally rearranging the clauses of rules until data flows strictly from left to right. The mode checking algorithm is extremely naive—we simply explore all rearrangements of the rule body until we find one that satisfies the dataflow constraints. Although this takes worst-case time factorial in the number of clauses, it seems to perform well enough in practice.

As Suppl is designed to be agnostic to the problem domain to which it is being applied, it is important that it be easy to extend the language with problem-specific programming facilities and to interface with an external system that generates the events and implements the actions returned by a Suppl program. In order to make this interface as easy as possible and to support the basic logic-programming facilities need for Suppl semantics, our runtime system for Suppl is based on the tuProlog system [11], a Prolog interpreter written in Java, which has a well-designed external function interface. To implement Suppl primitive functions and predicates simply requires writing a Java class containing methods with the correct names using the tuProlog's API, and arranging for the custom class to be loaded into the interpreter. The interpreter uses Java reflection to find the external functions and execute them as required.

[2] `http://web.cecs.pdx.edu/~rdockins/suppl/`

The executable part of SUPPL is deliberately quite similar to Prolog, and the mapping between SUPPL data structures and Prolog data structures is nearly trivial. The connection between the Java API and Prolog data structures is a little more distant, but the tuProlog API for manipulating Prolog terms is relatively easy to use. To get a flavor for the required interface programming, consider the following example file, which implements a simple primitive predicate named `primOp` at two different modes.

```
public class NewLibrary
  extends alice.tuprolog.Library {
  // in this case, the io mode implementation
  // also works for mode ii
  public boolean primOp_ii_2(Term arg1, Term arg2) {
    return primOp_io_2(arg1,arg2);
  }

  public boolean primOp_io_2(Term arg1, Term arg2) {
    arg1 = arg1.getTerm();

    // build some new term
    Struct x = new Struct("mkAsdf", arg1);

    // try to unify x with arg2
    return engine.unify(x, arg2);
  }
}
```

This Java code is sufficient to implement the following declared SUPPL primitive.

```
data asdf ::= mkAsdf(string).

primitive predicate primOp(string, asdf).
mode primOp(in,in).
mode primOp(in,out).
```

SUPPL data constructs, as well as action and event instances, are all represented directly as functor applications in Prolog; lists and numbers are handled natively by the Prolog system. Strings are interpreted in Prolog as atoms.

In the tuProlog API, the `Struct` class (a subclass of `Term`) represents atoms, lists and functor applications. Above, `new Struct("mkAsdf",arg1)` constructs a new Prolog functor instance with name `mkAsdf` and a single argument, represented by `arg1`. This maps directly onto a SUPPL term built using the `mkAsdf` data constructor. The class `Number` (also a `Term` subclass) is used to represent numeric values. Primitive SUPPL types can be represented by arbitrary Java objects. These objects will be passed around by reference inside the policy code; the runtime will make use only of basic Java `Object` methods, like `equals` and `hashCode`.

Using a Prolog interpreter in this way is a relatively heavyweight implementation strategy and will be unsuitable for applications requiring very frequent policy queries or which have tight real-time deadlines. So, it would almost certainly not be acceptable for, say, a firewall to query a Suppl policy every time a packet arrives; however, it may be acceptable to query the policy every time a new connection is opened.

Here is some sample code showing how to set up the Suppl runtime environment and load a custom library and interact with a loaded policy.

```
public static void main( String[] args )
  throws Exception {

  SupplEngine engine = RunPolicy.setupEngine();
  NewLibrary lib = new NewLibrary(engine);
  engine.loadLibrary(lib);
  RunPolicy.loadTheories(args, engine);

  Term[] evargs = new Term[] {
     new Struct( "string literal" ),
     new Int( 6 ) };
  Struct event = new Struct( "notification", evargs );

  List<Term> actions =
     RunPolicy.handleEvent(engine, event);

  for( Term t : actions ) {
    t = t.getTerm();
    System.out.println(t.toString());
  }
}
```

This sample code will load any compiled policy files given as command line arguments, feed a single synthetic event into the policy engine and print the resulting actions.

The result of this design is that it should be easy to integrate Suppl-defined policies into existing Java applications whenever policy questions can be organized into the event-condition-action paradigm. In addition, only modest changes to the Suppl compiler should be required to target other Prolog systems, which would allow Suppl policies to integrate with applications written in languages other than Java.

6 Related work

Here we survey existing work including both explicitly domain-neutral languages and languages that were designed for network security applications but can easily be generalized to broader domains.

Generic policy languages The Policy Description Language (PDL) [19] is similar in many ways to Suppl; it is based on the ECA policy paradigm, is influenced by logic programming ideas and is also designed with ease of analysis in mind. PDL has only one form of rule, which states that an event causes a particular action provided some condition holds. A significant difference from Suppl is that PDL lacks any explicit notion of state; instead, time-varying policies can be written using rules that match on *event sets* that can examine events that occurred in the past. SUPPL event handlers can only examine the current event, but data tables allow a principled way to record information for later examination.

Ponder [8] is a language for expressing security policies interacting at various levels of the hardware/software stack: network firewalls, databases, Java runtime security. Ponder's approach to specifying policy is quite different to ours; it has a strongly-developed object model for roles, groups, membership, etc. and syntax for manipulating these objects. In contrast, SUPPL builds in nothing except primitive base types, and instead relies on the user (or a library author) to build a model of the problem domain in question.

Modern business rules management systems such as JRules [5] and Drools [17] include languages for defining arbitrary production rule systems that can be integrated into Java applications. While production rules have a declarative flavor, rule actions can actually contain arbitrary imperative code, and chaining among rules can cause complicated and opaque control flow logic. SUPPL enforces a more disciplined separation between conditions and actions.

Network security The Authorization Specification Language (ASL) [16] is a language for expressing certain kinds of access control policies. Like SUPPL, it takes inspiration from logic programming constructs, and the primary act of programming in ASL involves writing various kinds of rules: authorization rules, access control rules, data integrity rules, etc. ASL allows users to express various kinds of conflict resolution metapolicies. ASL seems to lack any method for expressing stateful policies.

The Flow-based Management Language (FML) [15] is a declarative language for managing enterprise network configuration. An FML policy is expressed as a set of implication rules, based on nonrecursive DATALOG with negation. There is no internal notion of state. The language design is tailored to support efficient (linear time) evaluation. Conflicts can be resolved either by ordering rules or by assigning priorities to primitive actions.

Procera [24] is a domain specific language (embedded in Haskell) for expressing networking policy using the framework of functional reactive programming. In this framework, one defines a policy program (conceptually) as a time-varying function from an infinite stream of input events to a stream of output events. Aside from the quite different programming model, Procera's status as an embedded DSL makes it more difficult to build static analysis tools, as any analyses must be able to handle essentially all of the constructs of the host language, Haskell, a large general-purpose language in its own right.

Conflict detection and resolution Conflict resolution has been studied in the context of PDL [6]. The PDL conflict resolution system allows users to declare the conditions under which a conflict occurs. At runtime, conflicts can be handled in a number of different ways by writing conflict monitors. These may resolve conflicts by choosing actions with higher priorities, by canceling all effects of the event causing the conflict, etc. Policy monitors are not expressible in PDL itself, but must be defined externally. SUPPL avoids the tricky issue of conflict *resolution* by passing it off instead to the external system we already assume must exist. Instead, we have concentrated our efforts on building a system to help users discover potential conflicts in their policies statically.

Dunlop et al. [12] present a system for both detecting and dynamically resolving policy conflicts. In their system, policies are stated using operators of deontic logic—in particular, modal operators for permission, prohibition and obligations. They propose a number of strategies for resolving conflicts at runtime (explicit priority values, new policy overrides old, specific policy overrides general, etc.) and suggest that no one strategy is appropriate for all uses.

7 Conclusion

SUPPL is a programming language designed from the ground up for expressing and reasoning about event-condition-action policies over arbitrary domains. It combines the power and simplicity of pure logic programming, used for describing conditions, with the flexibility and familiarity of imperative programming, used to connect events to actions. The language has been implemented and integrated into several Java-based network security applications. We are actively working to apply it in additional domains.

Perhaps the most important benefit of having a dedicated language for authoring policies is the opportunity to apply sophisticated static analyses to detect errors before a policy is fielded. We have developed a prototype of one such analysis, which discovers conflicts caused by inconsistent actions, making essential use of an external logic solver. As future work, we plan to extend this prototype—in particular, by improving the quality of feedback from the external solver to the programmer—and to apply the same approach to other static analyses, such as liveness or functional correctness.

Acknowledgments

This work was supported by the 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.

References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Intl. Conf. on Computer Aided Verification (CAV). LNCS, vol. 6806, pp. 171–177. Springer (Jul 2011)
2. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: (2008), `http://alt-ergo.lri.fr/`, the Alt-Ergo automated theorem prover

3. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing Polymorphism in SMT solvers. In: Intl. Workshop on Satisfiability Modulo Theories (SMT). ACM International Conference Proceedings Series, vol. 367, pp. 1–5 (2008)
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: Workshop on Intermediate Verification Languages. pp. 53–64. Wrocław, Poland (August 2011)
5. Boyer, J., Mili, H.: Agile Business Rule Development. Springer (2011)
6. Chomicki, J., Lobo, J., Naqvi, S.: Conflict resolution using logic programming. IEEE Trans. on Knowl. and Data Eng. 15(1), 244–249 (Jan 2003)
7. Clark, K.L.: Negation as failure. Logic and Data Bases pp. 293–322 (1977)
8. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder policy specification language. In: Policies for Distributed Systems and Networks. LNCS, vol. 1995, pp. 18–38. Springer (2001)
9. DARPA: Safety On Untrusted Network Devices (SOUND) (2011), Mission-oriented Resilient Clouds (MRC) program: DARPA-BAA-11-55
10. Dayal, U., Hanson, E.N., Wisdom, J.: Active database systems. In: Modern Database Systems. ACM (1994)
11. Denti, E., Omicini, A., Ricci, A.: Multi-paradigm Java-Prolog integration in tuProlog. Sci. Comput. Program. 57(2), 217–250 (Aug 2005)
12. Dunlop, N., Indulska, J., Raymond, K.: Methods for conflict resolution in policy-based management systems. In: Intl. Conf. on Enterprise Distributed Object Computing. IEEE (2003)
13. Frazier, G., Duong, Q., Wellman, M.P., Petersen, E.: Incentivizing responsible networking via introduction-based routing. In: Intl. Conf. on Trust and Trustworthy Computing. pp. 277–293. TRUST'11, Springer-Verlag, Berlin, Heidelberg (2011)
14. Garcia de la Banda, M., Stuckey, P.J., Harvey, W., Marriott, K.: Mode checking in HAL. Conference on Computational Logic (2000)
15. Hinrichs, T.L., Gude, N.S., Casado, M., Mitchell, J.C., Shenker, S.: Practical declarative network management. In: Workshop on Research on Enterprise Networking. pp. 1–10. WREN '09, ACM (2009)
16. Jajodia, S., Samarati, P., Subrahmanian, V.S.: A logical language for expressing authorizations. In: IEEE Symp. on Security and Privacy. IEEE (1997)
17. JBoss Drools Team: Drools documentation (2014), `http://docs.jboss.org/drools/release/6.1.0.Final/drools-docs/html\_single`
18. Kowalski, R., Kuehner, D.: Linear resolution with selection function. Artificial Intelligence 2, 227–260 (1971)
19. Lobo, J., Bhatia, R., Naqvi, S.: A policy description language. In: AAAI Conf. on Artificial Intelligence. American Association for Artificial Intelligence (1999)
20. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. 17, 348–75 (1978)
21. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 337–340. Springer (2008)
22. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury: an efficient purely declarative logic programming language. Journal of Logic Programming 29(1-3), 17–64 (1996)
23. Trieu, A., Dockins, R., Tolmach, A.: Conflict analysis for Suppl (2014), in preperation
24. Voellmy, A., Kim, H., Feamster, N.: Procera: A language for high-level reactive network control. HotSDN (2012)