Combining Static Model Checking with Dynamic Enforcement using the Statecall Policy Language

Anil Madhavapeddy

Imperial College,
South Kensington, London SW1 2AZ, UK
a.madhavapeddy@imperial.ac.uk

Abstract. Internet protocols encapsulate a significant amount of state, making implementing the host software complex. In this paper, we define the Statecall Policy Language (SPL) which provides a usable middle ground between ad-hoc coding and formal reasoning. It enables programmers to embed automata in their code which can be statically model-checked using SPIN and dynamically enforced. The performance overheads are minimal, and the automata also provide higher-level debugging capabilities. We also describe some practical uses of SPL by describing the automata used in an SSH server written entirely in OCaml/SPL.

Constructing modern Internet servers is a difficult proposition, since the software must encapsulate a significant amount of state and deal with a variety of incoming packet types, complex configurations and versioning inconsistencies. Network applications are also expected to be liberal in interpreting received data packets and must reliably deal with timing and ordering issues arising from the "best-effort" nature of Internet data traffic.

Due to this complexity, mechanical verification techniques are very useful to guarantee safety, security and reliability properties. One mature formal method used to verify properties about systems is *model checking*. Software model-checking involves: (*i*) creating an abstract model of a complex application; (*ii*) validating this model against the application; and (*iii*) checking safety properties against the abstract model. To non-experts, steps (*i*) and (*ii*) are often the most daunting. How does one decide which aspects of the application to include in the abstract model? How does one determine whether the abstraction inadvertently "hides" critical bugs? If a counter-example is found, how does one determine whether this is a genuine bug or just a modeling artifact?

In this paper, we present the Statecall Policy Language (SPL) which simplifies the model specification and validation tasks with a view to making model checking more accessible to regular programmers. SPL is a high-level modelling language which enables developers to specify models in terms of allowable program events (e.g. valid sequences of received network packets). We have implemented a compiler that translates SPL into both PROMELA and a general-purpose programming language (e.g. OCaml). The generated PROMELA can be used with SPIN [1] in order to check static properties of the model. The OCaml code provides an executable model in the form of a *safety monitor*. A developer

can link this safety monitor against their application in order to *dynamically* ensure that the application's behaviour does not deviate from the model. If the safety monitor detects that the application has violated the model then it logs this event and terminates the application.

Although this technique simplifies model specification and validation it is, of course, not appropriate for all systems. For example, dynamically shutting down a fly-by-wire control system when a model violation is detected is not an option. However, we observe that there *is* a large class of applications where dynamic termination, while not desirable, is preferable to (say) a security breach. Melange [2] focusses on constructing correct, clean-room implementations of Internet applications using statically type-safe languages, and SPL delivers real benefits in this area. None of the major implementations of protocols such as HTTP (Apache), SMTP (Sendmail/Postfix), or DNS (BIND) are regularly model-checked by their development teams. All of them regularly suffer from serious security flaws ranging from low-level buffer overflows to subtle high-level protocol errors, some of which could have been caught by using model checking. In this paper, we use the Melange SSH [3] server as an example of how an application using SPL can be model-checked without sacrificing performance (§3.1) and enforcing critical security properties (§3.2) that are informally specified in the RFC documents.

There is no "perfect" way of specifying complex state machines, and the literature contains many different languages for this purpose (e.g. SDL [4], Estelle [5], Statemate [6], or Esterel [7]). In recognition of this, the SPL language is very specialised to expressing valid sequences of packets for Internet protocols and is translated into a more general intermediate "Control Flow Automaton" representation first proposed by Henzinger et al. [8]. The output code is generated from this graph, allowing for other state machine languages to be used in the future without requiring the backend code generators to be rewritten.

1 Statecall Policy Language

SPL is used to specify sequences of events which represent non-deterministic finite state automata. The automaton inputs are referred to as *statecalls*—these can represent any program events such as the transmission of receipt of network packets or the completion of some computation. The syntax of the language is written using a familiar 'C'-like syntax, with built-in support for non-deterministic choice operators in the style of Occam's ALT [9]. Statecalls are represented by capitalized identifiers, and SPL functions use lower-case identifiers. Semicolons are used to specify sequencing (e.g. `S1;S2` specifies that the statecall `S1` must occur before the statecall `S2`).

1.1 Case Study

Before specifying SPL more formally, we explain it via a simple case study—the UNIX `ping` utility which transmits and receives ICMP Echo requests and measures their latencies. A simple `ping` automaton with just 3 statecalls could be written as:

```
automaton ping() {
  Initialize;
  multiple (1..) {
   Transmit_Ping;
   Receive_Ping;
  }
}
```

This automaton guarantees that the statecalls must initially operate in the following order: `Initialize`, `Transmit_Ping`, and `Receive_Ping`. Since a realistic implementation of ping transmits and receives packets continuously, we also use the `multiple` keyword in our SPL specification. Using this automaton, the `ping` process can perform initialisation once, and then transmit and receive ping packets forever; an attempt to initialise more than once is not permitted. In a realistic network a ping response might never be received, and the non-deterministic `either`/`or` operator allows programmers to represent this scenario.

```
automaton ping() {
  Initialize;
  multiple (1..) {
    Transmit_Ping;
    either {
      Receive_Ping;
    } or {
      Timeout_Ping;
    };
  }
}
```

`ping` provides a number of command-line options that can modify the program behaviour. For example, `ping -c 10` requests that only 10 ICMP packets be sent in total, and `ping -w` specifies that we must never timeout, but wait forever for a ping reply. We represent these constraints by introducing *state variables* into SPL as follows:

```
automaton ping(int max_count, int count, bool can_timeout) {
  Initialize;
  during {
    count = 0;
    do {
      Transmit_Ping;
      either {
        Receive_Ping;
      } or (can_timeout) {
        Timeout_Ping;
      };
      count = count + 1;
    } until (count >= max_count);
  } handle {
    Sig_INFO;
```

```
    Print_Summary;
  };
}
```

Observe that the `either/or` constructs can be conditionally guarded in the style of Occam's ALT, and state variables can be assigned in an imperative style. A long-running `ping` process would need to receive UNIX signals at any point in its execution, take some action, and return to its original execution. Signal handlers are often a source of bugs due to their extremely asynchronous nature [10]—SPL provides a `during/handle` construct (used in the example above) which models them by permitting a state transition into alternative statement blocks during normal execution of an SPL specification.

Once we are satisfied that our SPL specification is of suitable granularity, the SPL compiler is run over it. The compiler outputs several targets: (*i*) a graphical visualisation using the Graphviz tool [11] as seen in Figure 1 for the example above; (*ii*) a non-deterministic model in the PROMELA language; and (*iii*) an executable model designed to be linked in with an application. The OCaml interface for the executable model is shown below:

```
exception Bad_statecall

type t = [ `Initialize | `Print_summary | `Receive_ping
  | `Sig_info | `Timeout_ping | `Transmit_ping ]

type s
val init : max_count:int -> count:int -> can_timeout:bool ->
  unit -> s
val tick : s -> t -> s
```

This code is linked in with the main `ping` application, and appropriate calls to initialize the automaton and invoke statecalls are inserted in the code. Crucially, we do not mandate a single style of invoking statecalls; instead the programmer can choose between automatic mechanisms (e.g. MPL [2] packet parsing code can automatically invoke statecalls when transmitting or receiving packets), language-assisted means (e.g. functional combinators, object inheritance, or pre-processors such as `cpp`), or even careful manual insertion in places where other methods are inconvenient.

1.2 Syntax and Typing Rules

The SPL syntax is presented in Figure 2 using an extended Backus-Naur Form [12]. We represent terminals as *term*, tokens as `token`, alternation with {*one* | *two*}, optional elements as [*optional*], elements which must repeat once or more as (*term*)+ and elements which may appear never or many times as (*term*)*.

SPL is a first order imperative language, extended from Cardelli's simple imperative language [13]. We distinguish between *commands* (without a return value) and *expressions* which do have a return value. Function and automaton names are distinct, and are considered commands. Function types are written

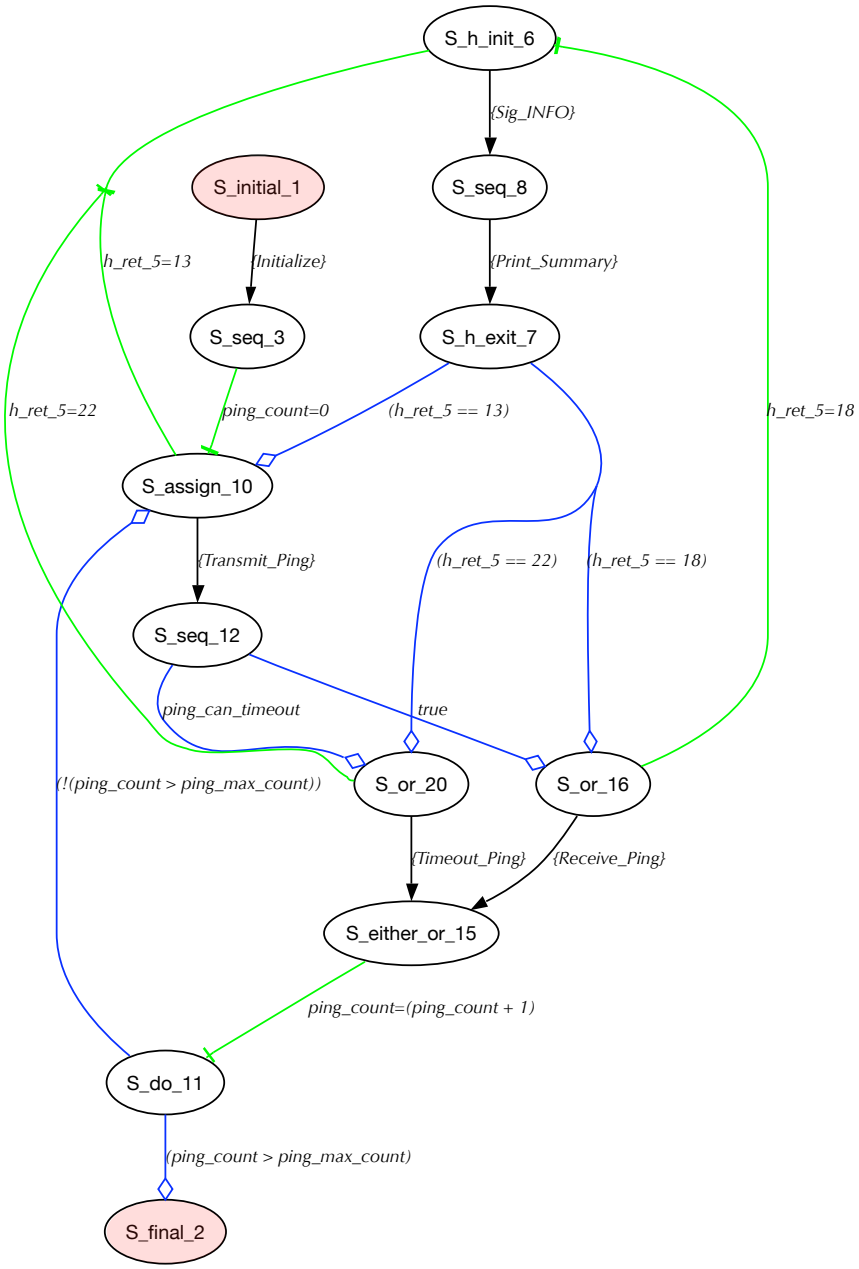

Fig. 1. Graph output of the example `ping` state machine. Red nodes indicate the start and final states, black edges are statecalls, blue edges are conditional, and green edges are state variable assignments

$\rho_1 \times \ldots \times \rho_i$, or abbreviated to $\boldsymbol{\rho}$. Γ_α represents a global environment with type signatures for functions and Γ a per-function environment containing state variable bindings. SPL does not have any built-in functions, so all type signatures are obtained from the SPL specifications.

Table 1 lists the imperative type judgements and Table 2 establishes the basic typing rules. Note that procedure environments contain only the variables passed in as arguments to the function declaration, and no global variables are permitted. Table 3 and Table 4 list the type rules for expressions and statements.

main → (*fdecl*)+ *eof*
fdecl → {`automaton` | `function`} *id* [*fargs*] *fbody*
fargs → `(` {`int` *id* | `bool` *id*} [`,` *fargs*] `)`
fcall-args → *id* [`,` *fcall-args*]
statecall-args → *statecall* [`,` *statecall-args*]
fbody → `{` (*statement*)* `}` [`;`]
int-range → `(` [*int*] `..` [*int*] `)` | `(` *int* `)`
statement → *statecall* `;` | *id* `(` *fcall-args* `)` `;`
| `always-allow` `(` *statecall-args* `)` *fbody*
| `multiple` *int-range fbody* | `optional` *fbody*
| `either` [*guard*] *fbody* (`or` [*guard*] *fbody*)+
| `do` *fbody* `until` *guard* `;`
| `while` *guard fbody*
| *id* `=` *expr* `;`
| `during` *fbody* (`handle` *fbody*)+
| `exit` `;` | `abort` `;`
guard → `(` *expr* `)`
expr → *int* | *id* | `(` *expr* `)`
| *expr* `+` *expr* | *expr* `-` *expr*
| *expr* `*` *expr* | *expr* `/` *expr*
| `-` *expr* | `true` | `false`
| *expr* `&&` *expr* | *expr* || *expr* | `not` *expr*
| *expr* > *expr* | *expr* >= *expr*
| *expr* < *expr* | *expr* <= *expr*
| *expr* = *expr*

Fig. 2. EBNF grammar for SPL specifications

2 Intermediate Representation

This section defines the Control Flow Automaton graph used as an intermediate representation of SPL specifications (§2.1), the semantics of multiple automata in the same SPL specification (§2.2), and finally optimisations applied to the CFA to reduce the number of states (§2.3). The CFA is a good abstraction for a software-based non-deterministic model and it is often used by model extraction tools (e.g. BLAST [8]) as the representation into which C source code is

Table 1. Type Judgments for SPL

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash C$	C is a well-formed command in Γ
$\Gamma \vdash E : A$	E is a well-formed expression of type A in Γ

Table 2. Basic environment and typing rules

$$\text{(Env } \phi\text{)} \quad \frac{}{\phi \vdash \diamond} \qquad \text{(Env x)} \quad \frac{\Gamma \vdash A \qquad I \notin dom(\Gamma)}{\Gamma, I : A \vdash \diamond} \qquad \text{(Type Int)} \quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Int}$$

$$\text{(Type Bool)} \quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Bool} \qquad \text{(Decl Proc)} \quad \frac{\phi, \boldsymbol{x} : \boldsymbol{\rho} \vdash C \qquad \Gamma_\alpha, I : \boldsymbol{\rho} \vdash \diamond}{\Gamma_\alpha \vdash (\textbf{fun } I \ (\boldsymbol{x} \times \boldsymbol{\rho}) = C)}$$

Table 3. Expression typing rules

$$\text{(Expr Bool)} \quad \frac{\Gamma \vdash \diamond \qquad x \in \{true, false\}}{\Gamma \vdash x : Bool} \qquad \text{(Expr Int)} \quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash N : Int} \qquad \text{(Expr Val)} \quad \frac{\Gamma_1, I : A, \Gamma_2 \vdash \diamond}{\Gamma_1, I : A, \Gamma_2 \vdash I : A}$$

$$\text{(Expr Not)} \quad \frac{\Gamma \vdash E_1 : Bool}{\Gamma \vdash \textbf{not } E_1 : Bool} \qquad \text{(Expr BoolOp)} \quad \frac{\Gamma \vdash E_1 : Bool \qquad \Gamma \vdash E_2 : Bool \qquad O_1 \in \{\textbf{and}, \textbf{or}\}}{\Gamma \vdash O_1(E_1, E_2) : Bool}$$

$$\text{(Expr IntOp)} \quad \frac{\Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int \qquad O_1 \in \{+, -, \times, \div\}}{\Gamma \vdash O_1(E_1, E_2) : Int}$$

$$\text{(Expr CompOp)} \quad \frac{\Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int \qquad O_1 \in \{=, >, \geq, <, \leq\}}{\Gamma \vdash O_1(E_1, E_2) : Bool}$$

Table 4. Command typing rules

$$\frac{\Gamma \vdash I : A \qquad \Gamma \vdash E : A}{\Gamma \vdash I \leftarrow E} \text{ (Cmd Assign)}$$

$$\frac{\Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash C_1;C_2} \text{ (Cmd Sequence)}$$

$$\frac{\Gamma \vdash C}{\Gamma \vdash \textbf{allow } C} \text{ (Cmd Allow)}$$

$$\frac{\Gamma \vdash C_{1..n} \qquad \Gamma \vdash E_{1..n} : Bool}{\Gamma \vdash \textbf{either } (C_1 \times E_1 \ldots C_n \times E_n)} \text{ (Cmd Either Or)}$$

$$\frac{\Gamma \vdash E : Bool \qquad \Gamma \vdash C}{\Gamma \vdash (\textbf{until } E = C)} \text{ (Cmd Do Until)}$$

$$\frac{\Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int \qquad \Gamma \vdash C}{\Gamma \vdash (\textbf{multiple } E_1\ E_2 = C)} \text{ (Cmd Multiple)}$$

$$\frac{\Gamma \vdash E : Bool \qquad \Gamma \vdash C}{\Gamma \vdash (\textbf{while } E = C)} \text{ (Cmd While)}$$

$$\frac{\Gamma^1_\alpha, I : \boldsymbol{\rho}, \Gamma^2_\alpha \vdash \diamond \qquad \Gamma \vdash \boldsymbol{x} : \boldsymbol{\rho}}{\Gamma^1_\alpha, I : \boldsymbol{\rho}, \Gamma^2_\alpha \vdash \textbf{call } I\ \boldsymbol{x}} \text{ (Cmd Function Call)}$$

$$\frac{}{\Gamma \vdash \textbf{exit}} \text{ (Cmd Exit)} \qquad \frac{}{\Gamma \vdash \textbf{abort}} \text{ (Cmd Abort)}$$

converted. Since there are a myriad of state-machine languages similar to SPL which share the properties formalised by Schneider's software automata [14], our adoption of the CFA representation ensures that the back-ends of the SPL toolchain (e.g. the Promela output) remain useful even if the front-end language is changed into something specialised for another task.

2.1 Control Flow Automaton

The SPL compiler transforms specifications into an extended Control Flow Automaton (CFA) [8] graph. A CFA represents program states and a finite set of state variables in blocks, with the edges containing conditionals, assignments, statecalls or termination operations. The CFA is non-deterministic and multiple states can be active simultaneously. More formally, our *extended control flow automaton* C is a tuple $(Q, q_0, X, S, Op, \rightarrow)$ where Q is a finite set of control locations, q_0 is the initial control location, X a finite set of typed variables, S a finite set of statecalls, Op a set of operations, and $\rightarrow \subseteq (Q \times Op \times Q)$ a finite set of edges labeled with operations. An edge (q, op, q') can be denoted $q \xrightarrow{op} q'$. The set Op of operations contains: (i) *basic blocks* of instructions, which consist of finite sequences of assignments **svar = exp** where **svar** is a state variable from X and **exp** is an equivalently typed expression over X; (ii) *conditional predicates* **if(p)**, where **p** is a boolean expression over X that must be true for the edge to be taken; (iii) *statecall predicates* **msg(s)**, where **s** is a statecall ($\mathbf{s} \in S$) received by the automaton; and (iv) *abort traps*, which immediately signal the termination of the automaton. From the perspective of a Mealy machine, the input alphabet Σ consists of statecall predicates and the output alphabet $\wedge$ is

the remaining operations. Thus a CFA graph is driven purely by statecall inputs, and the other types of operations serve to hide the state space explosion of a typical software model.

The CFA graph is constructed from SPL statements by recursively applying transformation rules to an initial state I and a final state O. Figure 3 illustrates the transformations for the basic SPL statements diagrammatically with the circles and lines representing CFA nodes and edges. The diamonds indicate a recursive application of the transformation rules with the initial and final states mapped to the input and outputs of the diamond node. Nodes within the dashed ellipses (named α, β, γ and so on) are newly created by the transformation rule. The **abort** and **exit** keywords signal the end of the automaton and thus do not connect to their output states. Each transformation rule has an environment ($\Gamma \times \Delta$) where Γ is the list of always allowed statecalls as seen in **allow** blocks and Δ represents statecalls which result in a transition to a handle clause (generated by the **during/handle** statement). A **during/handle** statement first creates all the handler nodes and transforms the main block with the handlers registered in the Δ environment. A **statecall** node creates a statecall edge and inserts appropriate edges to deal with **allow** and **during** handlers.

Some statements require the creation of new internal variables. The **multiple** call can optionally specify upper and lower bounds to the number of iterations; extra variables are automatically created to track these bounds in the CFA. **during/handle** statements create a new internal variable to track the state to which a handler must return. Function calls are either macro-expanded (if only called once) or temporary variables used to push and pop arguments in a single copy of the function graph (if called multiple times). An example of these internal variables can be seen in Figure 1 in our earlier `ping` sample.

2.2 Multiple Automata

It is often more convenient and readable to break down a complex protocol into smaller blocks which express the same protocol but with certain aspects factored out into simpler state machines. Accordingly, SPL specifications can define multiple automata, but the external interface hides this abstraction and only exposes a single, flat set of statecalls. The scope of automata names are global and flat; this is a deliberate design decision since the language is designed for light-weight abstractions that are embedded into portions of the main application code. Even a complex protocol such as SSH [3] can be broken down into smaller, more manageable automata—we have listed some of these in Appendix A. In this section, we explain how statecalls are routed to the individual automata contained in an SPL specification.

Each automaton executes in parallel and sees every statecall. If an automaton receives a statecall it was not expecting it reports an error. If *any* of the parallel automata report an error then the SPL model has been violated. When a statecall is received, it is dispatched *only* to automata which can potentially use that statecall at some stage in their execution.

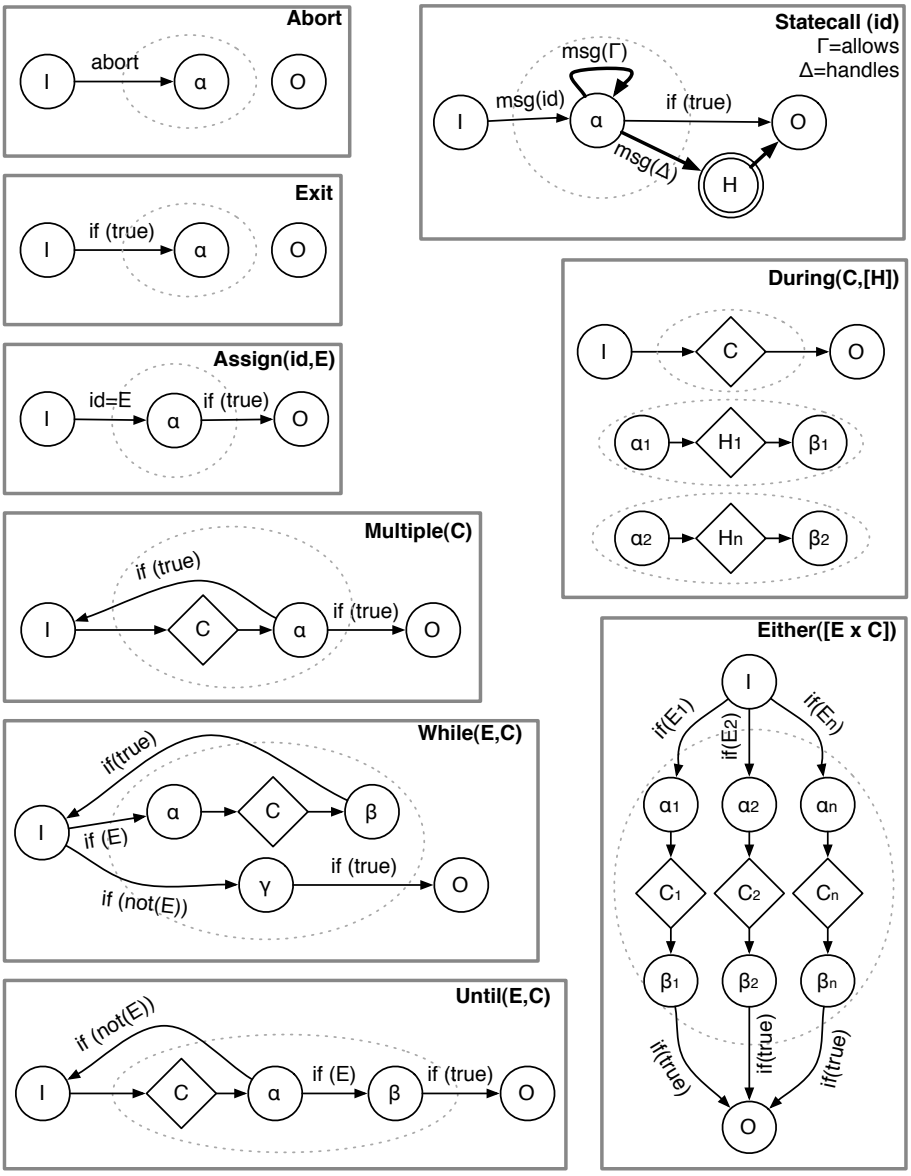

Fig. 3. Transformations of SPL statements into the corresponding CFA nodes

More formally, let $\mathcal{A}$ represent an automaton or function definition in an SPL specification. Let $\mathcal{V}(\mathcal{A})$ represent the union of all the statecalls referenced in $\mathcal{A}$, and $\mathcal{F}(\mathcal{A})$ be the list of all functions called from $\mathcal{A}$. The *potentially visible statecalls* $\mathcal{P}(\mathcal{A})$ are the set of statecalls which the automaton $\mathcal{A}$ will use at some stage in its execution where $\mathcal{P}(\mathcal{A}) = \mathcal{V}(\mathcal{A}) \cup \{\mathcal{P}(\mathcal{F}_0) \ldots \mathcal{P}(\mathcal{F}_n)\}$. A statecall is only dispatched to an automaton $\mathcal{A}$ if it is present in its potentially visible set $\mathcal{P}(\mathcal{A})$. Since the set of externally exposed statecalls $\mathcal{P}_{all} = \{\mathcal{P}(\mathcal{A}_0) \ldots \mathcal{P}(\mathcal{A}_n)\}$ is calculated by the union of all the potentially visible sets of the automata contained in an SPL specification, it trivially follows that every statecall will be dispatched to at least one automaton.

This mechanism allows complex protocols such as SSH to be broken down into simpler automata which are still connected together by common messages. The SPL compiler can output the list of statecalls which are shared between automata as a development aid; in practise while specifying Internet protocols we have observed that most automata share only one or two statecalls between them (normally global messages to indicate protocol termination or auth status).

2.3 Optimisation

The transformation rules defined earlier (§2.1) result in a CFA which has a number of redundant edges (e.g. *if(true)* conditionals). The SPL compiler reduces the number of states in the CFA without modifying the graph semantics. We iterate over the graph and perform constant folding [15] to simplify conditional expressions. Since SPL only has expressions with booleans and integers, the folding is a simple recursive pattern match.

The CFA is then traversed to eliminate redundant nodes: (i) for a node Q_i, all edges from the node are of the form $Q_i \xrightarrow{if(true)} Q_o$ or (ii) for a node Q_o, all edges pointing to the node are of the form $Q_i \xrightarrow{if(true)} Q_o$. The initial state of the automaton is left unoptimised, so that automata can have a single entry point for simplicity.

3 Compiler Outputs

The SPL compiler outputs automata in: (i) OCaml to be embedded as a dynamic safety monitor; (ii) Promela to statically verify safety properties using a model checker such as Spin; and (iii) HTML/AJAX to permit debugging of SPL models embedded in an executing application. Although we specifically describe an OCaml interface here, the compiler can also be easily extended to other type-safe languages (e.g. Java or C#), allowing application authors to write programs in their language of choice and still use the SPL tool-chain.

When describing each output, we will also analyze that output's use in the Melange SSH server. The Melange SSH server is written in pure OCaml, and uses the Meta Packet Language [2] to do the low-level packet parsing, and SPL to enforce the higher-level protocol constraints. The SSH protocol itself is defined in the form of Internet RFC documents [3].

3.1 OCaml

The OCaml output from the SPL compiler is designed to: (*i*) dynamically enforce the SPL model and raise an exception if it is violated; and (*ii*) provide real-time monitoring, debugging and logging of the SPL models. The SPL compiler generates OCaml code with a simple external interface which provides a: (*i*) variant type of statecalls for that model; (*ii*) constructor for a fresh automaton; and (*iii*) tick function which accepts a statecall and advances the automaton.

If a bad statecall is received, the automaton raises an exception. The interface is purely functional, thus allowing an automaton to be "rolled back" by keeping a list of previous automaton values.

The internal implementation takes several steps to make transitions as fast as possible. Since the only edges in the CFA which can "block" during execution are the statecall edges, all other edges are statically unrolled during compile-time code generation. When unrolling non-statecall edges during code generation, assignment operations are statically tracked by the SPL compiler in a symbol table. This permits the compiler to apply constant folding when the resultant expressions are used as part of conditional nodes (or when creating new state descriptors). Multiple conditional checks involving the same variable are grouped into a single pattern match (this is useful in SPL specs with **during/handle** clauses). These are necessary even when using the optimising OCaml compiler since they represent constraints present in the SPL specification which are difficult to spot in the more low-level OCaml code output.

The performance impact of several automata running in parallel in an application is minimal. In the Melange SSH server, written purely in OCaml and using SPL specifications to enforce constraints in the transport and connection layers of the protocol, it has around a 2-3% impact on the throughput of the server during bulk copy operations (see Figure 4). Some of the SPL specifications used in the SSH server are listed in Appendix A, with the full versions present in the Melange source code at http://melange.recoil.org/.

3.2 Model Checking

The SPL compiler also output Promela models from the SPL input, providing an easy way to statically reason about properties which are then dynamically enforced by the OCaml run-time automata. In the case of the SSH protocol, the SPL specification for the transport, authentication and global channel handling is a complex state machine, and an exhaustive safety verification in Spin without any additional LTL constraints (i.e. testing assertions and invalid end-states) requires around 400MB of RAM and one minute to verify on a dual-G5 1.25GHz PowerMac. Spin reports the following statistics:

```
State-vector 48 byte, depth reached 78709, errors: 0
1.41264e+07 states, stored (1.99736e+07 visited)
2.59918e+07 states, matched
4.59654e+07 transitions (= visited+matched)
7.85945e+07 atomic steps
```

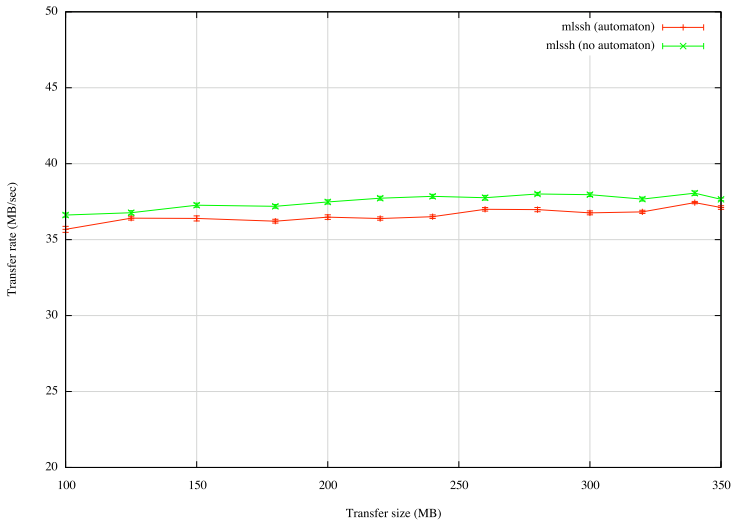

Fig. 4. Performance of the OCaml SSH server with and without the SPL automata

The large number of atomic steps show the complexity reduction which results from the SPL compiler inserting `atomic` statements in the generated PROMELA to simulate the execution semantics of the OCaml safety monitors. Before this optimisation, messages would unnecessarily be interleaved and verification took orders of magnitude longer.

We now list some of the LTL formulae applied to the PROMELA output of the SSH global automaton and describe the security properties which they enforce. Unlike some other tools which translate state machine languages into PROMELA (e.g. Scott's SWIL language for interface descriptions [16]), we never require the manual modification of the PROMELA code (which would be dangerous since the equivalence between the model and the dynamically enforced SPL automaton would not be guaranteed any more). Instead, globally allocated state variables[1] are exposed within the model which can be referenced with LTL formulae, as shown below:

- $\Box(\texttt{a} \rightarrow \Box\texttt{a})$ where $(\texttt{a} \leftarrow \texttt{transport\_encrypted})$ which informally reads "once the transport is encrypted, it will remain encrypted". This check ensures that the transport layer can never turn off encryption once a secure transport has been established for the lifetime of that connection.
- $\Box(\texttt{a} \rightarrow \Box(\texttt{a} \;\&\&\; \texttt{b}))$ where $(\texttt{a} \leftarrow \texttt{transport\_serv\_auth})$ and $(\texttt{b} \leftarrow \texttt{transport\_encrypted})$ which informally reads "in the `transport` automaton, once `serv_auth` is `true`, both `serv_auth` and `encrypted` remain

[1] SPIN does not support partial order evaluation over local variables, so the SPL compiler safely promotes automaton-local variables to a global scope.

`true` forever". This guarantees that authentication can only happen over an encrypted connection.
- $\Box$`a` where (`a` $\leftarrow$ `auth_success` + `auth_failed` < 2) informally reads "in the `auth` automaton, `success` and `failure` must never simultaneously be `true`". This restriction lets us use two boolean variables instead of a larger integer to store the 3 values for undecided, success or failure authentication states.
- $\Box$(`a` $\rightarrow X$(`b` $\mid\mid \Box\Diamond$`c`)) where (`a` $\leftarrow$ `p` == `Transmit_Auth_Success`) and (`b` $\leftarrow$ `auth_success`) and (`c` $\leftarrow$ `err`) informally reads "when an authentication success packet is transmitted, it must immediately be followed by the `success` variable being `true` or always eventually lead to an error."
- $\Box$(`a` $\rightarrow$ (`b` $\mid\mid \Box\Diamond$`c`)) where (`a` $\leftarrow$ `p` == `Transmit_Transport_Accept_Auth`) and (`b` $\leftarrow$ `transport_encrypted`) and (`c` $\leftarrow$ `err`) which informally reads "if the authentication service is unlocked then the transport layer must be encrypted or an error always eventually occurs". This matches the security considerations section of the SSH authentication specification in RFC4252 [17] which states that "it assumed *(sic)* that this runs over a secure transport layer protocol, which has already authenticated the server machine, established an encrypted communications channel [...]".
- $\Box$(`a` $\rightarrow$ (`b` $\mid\mid \Box\Diamond$`c`)) where (`a` $\leftarrow$ `p` == `Receive_Channel_Open_Session`) and (`b` $\leftarrow$ `auth_success`) and (`c` $\leftarrow$ `err`) which informally reads "requests to open a new channel are only allowed when authentication has been successful, or an error state is always eventually reached". This is in line with the security considerations section of the SSH connection specification in RFC4254 [18] which states that "this protocol is assumed to run on top of a secure, authenticated transport".

These properties all reflect restrictions expressed informally in the SSH specifications [3, 17, 18], and can now be sure to either work correctly in the running SSH server, or terminate the connection to prevent a potential security breach.

3.3 AJAX Debugging

The SPL compiler can also include debugging stubs in the executable automata, most usefully in the form of HTML/AJAX code which can be accessed via a web browser. This page contains a real-time graphical view of all the automata embedded in the program, along with the set of valid states they can transition to next. Since the granularity of the SPL automata are chosen by the programmer, this is much more useful than the "raw" models obtained through static code analysis which often include a lot of superfluous information.

Figure 5 shows a screen capture of the SPL AJAX debugger single-stepping through the global SPL automaton for the Melange SSH server. The MLSSH server is blocked waiting for password authentication, having previously attempted to authenticate via null and public-key authentication. In our experience, the debugger was a valuable tool to debug complex protocol bugs in our implementation, as the single-stepping view via this debugger is significantly higher level than the alternative provided by either the native OCaml debugger or `gdb`.

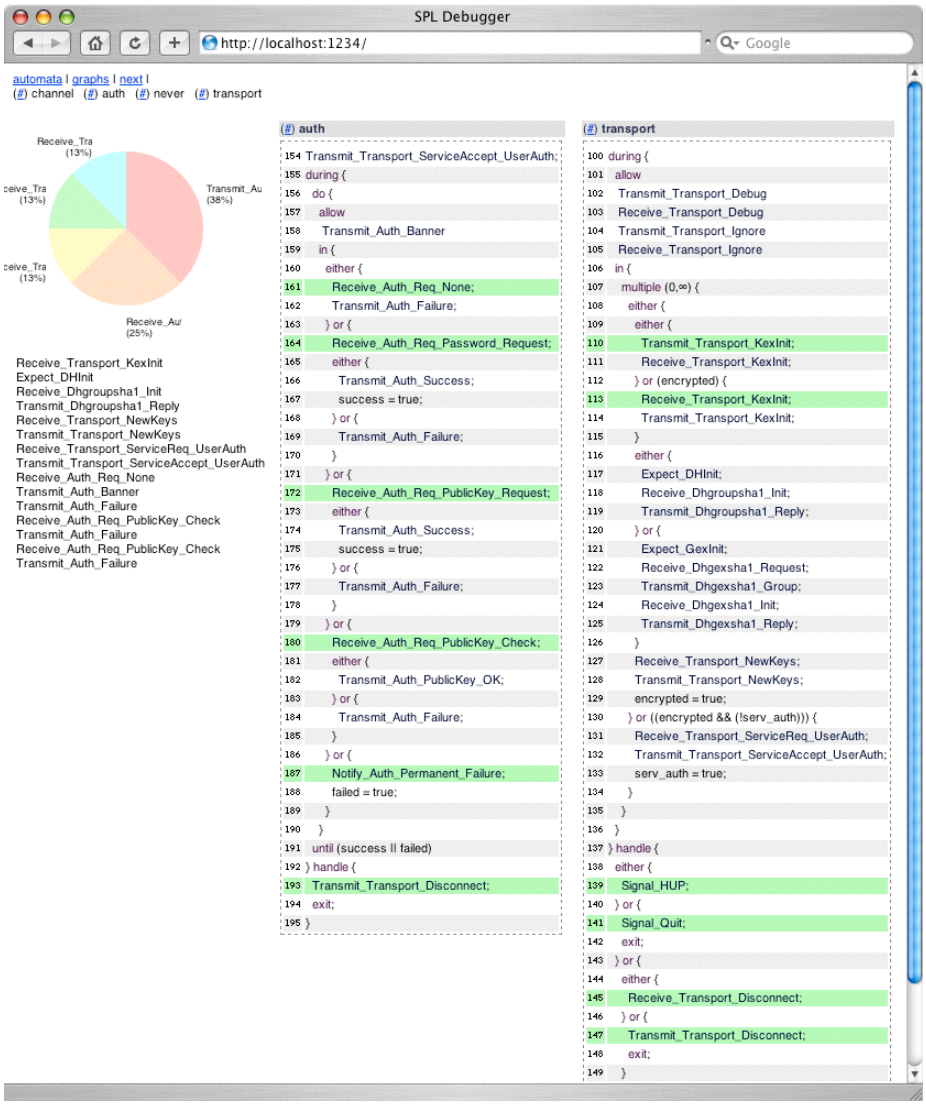

Fig. 5. Screen capture of the AJAX debugger embedded into the SSH daemon, showing the global SPL automaton. The green states are valid statecalls, the pie chart shows the 5 most popular statecalls in real time, and the list on the left show recent statecalls.

4 Related Work

The Bandera tool-chain [19] is designed to ease the model-checking of Java source code. It includes components for program analysis and slicing, transformation, and visualisation. Bandera accepts Java source as input and requirements written in the Bandera Specification Language (BSL) [20]. A key design goal of BSL is to hide the intricacies of temporal logic by placing emphasis on common specification coding patterns (e.g. pre- and post-conditions to functions). BSL is also strongly tied to the source program code via references to variables and methods names. Much of Bandera's utility arises from its tools for model construction which eliminate redundant components [21], simplifying the eventual output.

The BLAST [8] project introduced the *lazy abstraction* paradigm for verifying safety properties about systems code. Lazy abstractions follows the following steps: (*i*) an abstraction is extracted from the source code; (*ii*) the abstraction is model-checked; and (*iii*) the model is then refined using counter-example analysis. The process is repeated until the model is sufficiently refined, and the resulting proof certificates are based on Proof Carrying Code [22]. This mechanism helps make the model extraction process more scalable by reducing the amount of time and effort required to create abstractions of systems code. In contrast to the conventional abtract-verify-refine loop, lazy abstraction builds abstract models on demand from the original source code. This results in a non-uniformedly detailed model which contains just enough detail to show a counter-example to the developer. SPL also provides an alternative way to provide non-uniform models by permitting the programmer to choose the level of granularity they want to write the models in.

Alur and Wang have tackled the problem of model checking real-world protocols by extracting a specification from RFCs and using symbolic refinement checking to verify the model against protocol implementations written in C [23]. They evaluate their technique by creating and verifying models of DHCP and PPP, and conclude that "*[manual model extraction] is unavoidable for extracting specification models since RFC documents typically describe the protocols in a tabular, but informal, format*".

The Model-Carrying Code (MCC) project led by Sekar combines the model-extraction techniques described earlier with system call interception to provide a platform for the safe execution of untrusted binaries [24]. Untrusted code is bundled with a model of its security-relevant behaviour which can be formally verified against a local security policy by a model checker. The execution of the binary is dynamically verified by syscall interception to fit the model and the application terminated if a violation occurs. As Wagner and Soto point out [25], the low-level nature of syscall interception makes it easy for attackers to launch an observationally equivalent attack by crafting a valid sequence of syscalls, and so this technique is only useful as a last-resort if more formal and reliable verification techniques against the source code cannot be applied. We have drawn inspiration from the work described above, in particular the MCC approach of providing static models and dynamic enforcement, but our work operates at a higher level with explicit support from the application source code.

5 Conclusions

We have described the Statecall Policy language, which aims to provide a usable mechanism for programmers to integrate lightweight models into complex networked software. We solve the code/model equivalence problem by specifying models in our SPL language, and compiling them to multiple outputs for different purposes—model checking using SPIN by outputting PROMELA code, dynamical enforcement executables in OCaml, and even HTML/AJAX stubs for run-time debugging. It is currently targeted at applications written in OCaml and model checked using SPIN, but is simple to port to other languages and tools due to its use of the Control Flow Automaton intermediate graph.

We have also described practical uses of SPL in our complex Secure Shell server which uses several complex models to enforce critical security properties that are only informally specified in the official SSH RFCs.

We gratefully acknowledge funding from Intel Research and the UK Engineering and Physical Sciences Research Council grant `EP/F024037/1`.

References

1. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley (September 2003)
2. Madhavapeddy, A., Ho, A., Deegan, T., Scott, D., Sohan, R.: Melange: creating a "functional" internet. In Ferreira, P., Gross, T.R., Veiga, L., eds.: EuroSys, ACM (2007) 101–114
3. Ylonen, T., Lonvick, C.: The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard) (January 2006)
4. SDL: SDL forum society. Technical Report Recommendation Z.100, International Telecommunications Union, Geneva (1993)
5. ISO: Estelle—a formal description technique based on an extended state transition model. ISO 9074, International Organisation for Standardization, Geneva (1997)
6. Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., a. Shtul-Trauring: Statemate: a working environment for the development of complex reactive systems. In: Proceedings of the 10th International Conference on Software Engineering (ICSE), Los Alamitos, CA, USA, IEEE Computer Society Press (1988) 396–406
7. Berry, G.: III. In: The Foundations of Esterel: Proof, Language, and Interaction (Essay in Honor of Robin Milner). MIT Press (May 2000) 425–454
8. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Proceedings of the 14th International Conference on Computer Aided Verification (CAV), London, UK, Springer-Verlag (2002) 526–538
9. Jones, G.: Programming in Occam. Prentice-Hall, Hertfordshire, United Kingdom (1986)
10. Chen, H., Wagner, D.: MOPS: an infrastructure for examining security properties of software. In: Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS), New York, NY, USA, ACM Press (2002) 235–244
11. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software—Practice and Experience **30**(11) (2000) 1203–1233

12. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised report on the algorithm language ALGOL 60. Communications of the ACM **6**(1) (1963) 1–17
13. Cardelli, L.: Type systems. In Tucker, A.B., ed.: The Computer Science and Engineering Handbook. CRC Press (1997) 2208–2236
14. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information Systems Security **3**(1) (2000) 30–50
15. Aho, A.V., Ullman, J.D.: Principles of Compiler Design. Computer Science and Information Processing. Addison-Wesley, Reading, MA, USA (August 1977)
16. Scott, D.J.: Abstracting Application-Level Security Policy for Ubiquitous Computing. PhD thesis, University of Cambridge (2005)
17. Ylonen, T., Lonvick, C.: The Secure Shell (SSH) Authentication Protocol. RFC 4252 (Proposed Standard) (January 2006)
18. Ylonen, T., Lonvick, C.: The Secure Shell (SSH) Connection Protocol. RFC 4254 (Proposed Standard) (January 2006)
19. Corbett, J.C., Dwyer, M.B., Hatcliff, J.: A language framework for expressing checkable properties of dynamic software. In Havelund, K., Penix, J., Visser, W., eds.: Proceedings of the SPIN Software Model Checking Workshop. Volume 1885 of Lecture Notes in Computer Science., Springer (2000) 205–223
20. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby: Expressing checkable properties of dynamic systems: the Bandera Specification Language. International Journal on Software Tools for Technology Transfer **4**(1) (2002) 34–56
21. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from Java source code. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE), New York, NY, USA, ACM Press (2000) 439–448
22. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France, ACM Press (1997) 106–119
23. Alur, R., Wang, B.Y.: Verifying network protocol implementations by symbolic refinement checking. In: Proceedings of the 13th International Conference on Computer Aided Verification (CAV), London, UK, Springer-Verlag (2001) 169–181
24. Sekar, R., Venkatakrishnan, V., Basu, S., Bhatkar, S., DuVarney, D.C.: Model-carrying code: a practical approach for safe execution of untrusted applications. In: Proceedings of the Nineteenth ACM symposium on Operating Systems Principles, New York, NY, USA, ACM Press (2003) 15–28
25. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In Atluri, V., ed.: Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS), ACM (August 2002) 255–264

A SPL Policies for Secure Shell

In this appendix, we list an excerpt of the SPL policies for the Secure Shell (SSH) protocol. The full policies may be found in the Melange source code. There are two automata listed here which run in parallel (§2.2) and represent the transport and authentication layers respectively. The transport layer establishes an encrypted connection, and the authentication layer handles the negotiation of user credentials.

```
automaton transport (bool encrypted, bool serv_auth) {
  during {
    always_allow (Transmit_Transport_Debug,
        Receive_Transport_Debug, Transmit_Transport_Ignore,
        Receive_Transport_Ignore) {
      multiple {
        either {
          either {
            Transmit_Transport_KexInit;
            Receive_Transport_KexInit;
          } or (encrypted) {
            Receive_Transport_KexInit;
            Transmit_Transport_KexInit;
          }
          either {
            Expect_DHInit;
            Receive_Dhgroupsha1_Init;
            Transmit_Dhgroupsha1_Reply;
          } or {
            Expect_GexInit;
            Receive_Dhgexsha1_Request;
            Transmit_Dhgexsha1_Group;
            Receive_Dhgexsha1_Init;
            Transmit_Dhgexsha1_Reply;
          }
          Receive_Transport_NewKeys;
          Transmit_Transport_NewKeys;
          encrypted = true;
        } or (encrypted && !serv_auth) {
          Receive_Transport_ServiceReq_UserAuth;
          Transmit_Transport_ServiceAccept_UserAuth;
          serv_auth = true;
        }
      }
    }
  } handle {
    either { Signal_HUP; }
    or {
      either { Receive_Transport_Disconnect; }
      or {
        optional { Signal_QUIT; }
        Transmit_Transport_Disconnect;
        exit;
      }
    } or { Receive_Transport_Unimplemented; }
  }
}
```

```
automaton auth (bool success, bool failed) {
  Transmit_Transport_ServiceAccept_UserAuth;
  during {
    do {
      always_allow (Transmit_Auth_Banner) {
        either {
          Receive_Auth_Req_None;
          Transmit_Auth_Failure;
        } or {
          Receive_Auth_Req_Password_Request;
          either {
            Transmit_Auth_Success;
            success = true;
          } or {
            Transmit_Auth_Failure;
          }
        } or {
          Receive_Auth_Req_PublicKey_Request;
          either {
            Transmit_Auth_Success;
            success = true;
          } or {
            Transmit_Auth_Failure;
          }
        } or {
          Receive_Auth_Req_PublicKey_Check;
          either {
            Transmit_Auth_PublicKey_OK;
          } or {
            Transmit_Auth_Failure;
          }
        } or {
          Notify_Auth_Permanent_Failure;
          failed = true;
        }
      }
    } until (success || failed);
  } handle {
    Transmit_Transport_Disconnect;
    exit;
  }
}
```