

# Verifying a Signature Architecture — A Comparative Case Study

David Basin<sup>1</sup>, Hironobu Kuruma<sup>2</sup>, Kunihiro Miyazaki<sup>2</sup>, Kazuo Takaragi<sup>2</sup>, Burkhart Wolff<sup>1</sup>

<sup>1</sup>ETH Zurich, Zurich, Switzerland

<sup>2</sup>Hitachi Systems Development Laboratory, Yokohama, Japan

**Abstract.** We report on a case study in applying different formal methods to model and verify an architecture for administrating digital signatures. The architecture comprises several concurrently executing systems that authenticate users and generate and store digital signatures by passing security relevant data through a tightly controlled interface. The architecture is interesting from a formal-methods perspective as it involves complex operations on data as well as process coordination and hence is a candidate for both data-oriented and process-oriented formal methods.

We have built and verified two models of the signature architecture using two representative formal methods. In the first, we specify a data model of the architecture in Z that we extend to a trace model and interactively verify by theorem proving. In the second, we model the architecture as a system of communicating processes, which we verify by finite-state model checking. We provide a detailed comparison of these two different approaches to formalization (infinite state with rich data types versus finite state) and verification (theorem proving versus model checking). Contrary to common belief, our case study suggests that Z is well suited for temporal reasoning about process models with rich data. Moreover, our comparison highlights the advantages of proving theorems about richer models and provides evidence that, in the hands of an experienced user, theorem proving may be neither substantially more time-consuming nor more complex than model checking.

**Keywords:** Formal methods; Comparison; Theorem proving; Model checking; Security; Case study

**Note to reviewers:** All definitions and proof scripts for this case study are available on the web at [kisogawa.inf.ethz.ch/WebBIB/publications/papers/2005/HSD.pdf](http://kisogawa.inf.ethz.ch/WebBIB/publications/papers/2005/HSD.pdf) and [people.inf.ethz.ch/basin/spin-models.tar](http://people.inf.ethz.ch/basin/spin-models.tar). The former contains the HOL-Z formalization and the latter the PROMELA formalization.

## 1. Introduction

While there is increasing consensus about the usefulness of formal methods for developing and verifying critical systems, there are many options and schools of thought on how best to do this. Formal methods can be loosely characterized along different dimensions in terms of what views of a system they emphasize, the proof techniques used, etc. When most of the system's complexity stems from the way that processes interact, and the data manipulations are comparatively simple, then the use of a process-oriented modeling

language, like a process algebra or some kind of communicating automata, is typically favored and model checking is the preferred means of verification. On the other hand, when system data is structured into rich data types (e.g., formalizing problem domains, interface requirements, and the like) that are subject to complex manipulations, then data-oriented modeling languages are considered superior and verification is carried out by theorem proving. But what about systems whose design encompasses both complex data and nontrivial interaction and whose requirements speak about both the operations on data and their temporal ordering? Here there is less consensus and the options available include using abstraction to simplify the data model to enable model checking, theorem proving, and even combining formal methods.

In this paper, we look at an example of one such system: a security architecture used for a digital signature application. The architecture is based on the secure operating system DARMA (Hitachi’s platform for Dependable Autonomous hard Realtime Management) [ASS<sup>+</sup>99], which is used to control the interaction between different subsystems, running on different operating platforms. In particular, DARMA is used to ensure data integrity by separating user API functions, which run on a potentially open system (e.g., connected to the Internet), from those that actually manipulate signature-relevant data, which run on a separate, protected system. Any model of this architecture must formalize both the processes that run on the different platforms and the data that they manipulate to produce signatures. Moreover, the modeling formalism must be capable of formalizing data-integrity requirements, expressed as temporal properties about how the different data stores should change.

We present two models of the signature architecture: one based on a data model formalized using Z [Spi92], and the other as a system of communicating processes formalized in PROMELA, the input language to the Spin model checker [Hol04]. In both languages, we are faced with the question of how to model the two aspects — rich data combined with process interaction — and verify the resulting models. In the Z model, we formalize a classical data model that describes the architecture in terms of component states and state transitions. Afterwards, we exploit the fact that Z is a very rich specification language, and we extend the data model to a simple process model that describes the system’s semantics in terms of the set of its traces. This provides a basis for naturally formalizing the system’s security requirements as trace requirements. The main challenge then is verification, which requires interactively establishing invariants by induction over the set of system traces. In contrast, in our PROMELA model we focus on processes and their interactions. To verify the resulting model automatically, we simplify the data model such that the resulting system is finite state. The main challenges here are making this simplification, formalizing the properties that the resulting model should fulfill, and managing the complexity of model checking. Our case study provides concrete examples of these problems and how we have handled them.

**Contributions** Our main contribution is to provide a detailed comparison of these two different approaches to formalization (infinite state with rich data types versus finite state and process-oriented) and verification (theorem proving versus model checking). Our account is both quantitative and qualitative and sheds light on the relative strengths and weaknesses of the two approaches. Perhaps surprisingly, our experience in this case study is that, in the hands of an experienced user, theorem proving is neither substantially more time-consuming nor more complex, and in some regards it is considerably simpler, than working with a process-oriented view alone using a model checker. Moreover, we document a number of tradeoffs where the additional complexity is counterbalanced by additional benefits, for example, a more general architecture, stronger theorems, and an increased confidence in the system gained by formalizing and proving system invariants.

Our second contribution concerns the suitability of Z for formalizing process-oriented models and requirements. We show here how the use of a sufficiently expressive data-modeling language provides a foundation for formalizing a trace-based model of process interaction. The ideas here are general and should carry over to other ways of composing processes. It follows that there is no need to resort to different formal methods to formalize and combine the different system views since this can all be done within Z itself. The practical benefit of this is not only the simplicity of a single formal method, but also the direct use of general-purpose tools. An example of this is the HOL-Z environment for the Isabelle theorem prover, whose use we describe in this paper.

**Organization** In Section 2, we provide an informal overview of both the signature architecture and its security requirements. In Sections 3 and 4, we describe how we used Isabelle/HOL-Z and Spin, respectively, to model and verify the architecture. In Section 5, we compare approaches and in Section 6 we discuss related work and draw conclusions.

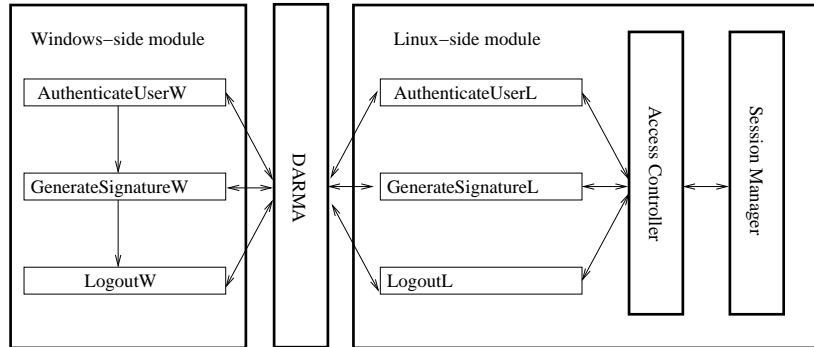


Fig. 1. The Signature Architecture

Note that in the interest of brevity, only illustrative aspects of the models and proofs are presented. All definitions and complete proof scripts for the two case studies are given in [BKTW04] and [Pro05].

## 2. The Signature Architecture

### 2.1. Overview

The signature architecture is based on two ideas. The first is that of a *hysteresis signature* [SM02], which is a cryptographic approach designed to overcome the problem that, for some applications, digital signatures should be valid for very long time periods. Hysteresis signatures address this problem by chaining signatures together so that the signature for each document signed depends on hash values computed from all previously signed documents. These chained signatures constitute a signature log and to forge even one signature in the log an attacker must forge (breaking the cryptographic functions behind) a chain of signatures.

The signature system reads the private keys of users from key stores, and reads and updates signature logs. Hence, the system’s security relies on the confidentiality and integrity of this data. The second idea is to protect these using a secure operating platform. For this purpose, Hitachi’s DARMA system [ASS<sup>+</sup>99] is used to separate the user’s operating system (in practice, Windows) from a second operating system used to manage system data (e.g., Linux). This compartmentalization plays a role analogous to network firewalls, but here the two systems are protected by controlling how functions in one system can call functions in the other. In this way, one can precisely limit how users access the functions and data for hysteresis signatures that reside in the Linux operating system space.

Our model is based on a 13 page Hitachi document, which describes the signature architecture using diagrams (like Figures 1 and 2) and text, as well as discussions with Hitachi engineers.

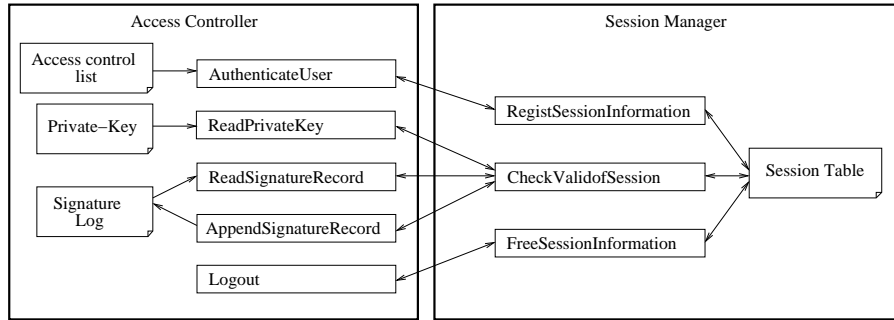
### 2.2. Functional Units and Dataflow

The signature architecture is organized into five modules, whose high-level structure is depicted in Figure 1. The thick-lined boxes represent modules and the thin-lined boxes represent individual functions.

The first module contains three functions, which execute in the operating system space of the user. We call this the “Windows-side module” to reflect the (likely) scenario that they are part of an API available to programs running under the Windows operating system. These functions are essentially proxies. When called, they forward their parameters over the DARMA module to the corresponding functions in the second, protected system, which is here called the “Linux-side module”, again reflecting a likely implementation. There are two additional (sub)modules, each also executing on the second system, which package data and functions for managing access control and sessions.

To create a hysteresis signature, a user takes the following steps on the Windows side:

1. The user application calls *AuthenticateUserW* to authenticate the user and generate a session identifier.
2. The application calls *GenerateSignatureW* to generate a hysteresis signature.



**Fig. 2.** The Access Controller and Session Manger Modules

#### Parameters

##### Input:

*username*: Name of the user who generates the hysteresis signature.

*password*: The *password* for *username*

##### Output:

*SessionID*: If the user authentication is successful,  $SessionID > 0$ , otherwise  $SessionID \leq 0$ .

#### Details

1. Sends *username*, *password* and *command* to Linux side using *CommunicateW*. The *command* is information used by the Linux-side module to distinguish the type of data that it receives.
2. Outputs *SessionID* returned by *CommunicateW*.

**Fig. 3.** Interface Description for *AuthenticateUserW*

3. The application calls *LogoutW* to logout, ending the session.

As explained above, each of these functions uses DARMA to call the corresponding function on the Linux side and DARMA serves to restrict access from the Windows side to only these three functions. The Linux functions themselves may call any other Linux functions, including those of the *Access Controller*, which controls access to data (private keys, signature logs, and access control lists). The *Access Controller* in turn uses functions provided by the *Session Manager*, which manages session information (*SessionID*, etc.), as depicted in Figure 2.

The Hitachi documentation provides an interface description for each of these functions. Two representative examples are presented in Figures 3 and 4. These are the descriptions of the functions *AuthenticateUserW* and *AuthenticateUserL*. The former calls DARMA and returns a session identifier while the latter does the actual work of checking the password and communicating with the access controller.

#### Parameters

##### Input:

*username*: Sent by *AuthenticateUserW* through *Darma*.

*password*: Sent by *AuthenticateUserW* through *Darma*.

##### Output:

*SessionID*: If the user authentication is successful, then  $SessionID > 0$ , otherwise  $SessionID \leq 0$ .

#### Details

1. Calculate the hash value of *password* using the Keymate/Crypto API. If successful, go to step 2, otherwise set *SessionID* to *CryptErr* ( $\leq 0$ ) and return.
2. Authenticate the user using the function *AuthenticateUser* of *Access Controller*.
3. Output *SessionID* returned by *AuthenticateUser*.

**Fig. 4.** Interface Description for *AuthenticateUserL*

### 2.3. Properties

The Hitachi documentation also states three requirements that the signature architecture should fulfill. These state that authenticated users are limited to generating one signature (with their private key) per authentication.

- R1.** The signature architecture must authenticate a user before the user generates a hysteresis signature.
- R2.** The signature architecture shall generate a hysteresis signature using the private key of an authenticated user.
- R3.** The signature architecture must generate only one hysteresis signature per authentication.

We will subsequently see how to model these requirements as properties of traces in both  $Z$  and linear temporal logic.

## 3. Modeling and Verification with Isabelle/HOL-Z

### 3.1. HOL-Z

For our first model, we used  $Z$  as our modeling language and the HOL-Z environment for theorem proving. As  $Z$  is well established and extensively documented, e.g., [Int, Spi92, WD96], we will assume that the reader has basic familiarity with it. HOL-Z [BRW03] is an environment built upon the Isabelle/HOL system [NPW02]. The HOL-Z environment provides a front end for creating “literate specifications”, where specifications are mixed with informal explanations and are constructed as  $\text{\LaTeX}$  documents, typeset using standard  $Z$  macros and idioms. These specifications are processed by HOL-Z and translated into a conservative shallow embedding of  $Z$  in HOL. HOL-Z also provides tactic support tailored to reasoning about  $Z$  specifications and implements various verification and refinement techniques.

### 3.2. The Data Model

Our formalization of the signature architecture’s state and operations is standard and closely follows Hitachi’s informal specification. We formalize a state schema for each of the different system modules and an operation schema for each function, based on their informal description.

**State Schemas.** As examples, we present two state schemas: the session manager and DARMA. The session manager maintains a session table (*session\_table*) and the set of active session identifiers (*session\_IDs*), i.e. those session identifiers currently in use. A session tables associates with user names and session identifiers information on access permissions for keys and the signature log (omitted here).

$$\begin{aligned} \text{SESSION\_TABLE} = &= \\ & (\text{USER\_ID} \setminus \{\text{NO\_USER}\}) \leftrightarrow \\ & (\text{SESSION\_ID} \setminus \text{AUTH\_ERRORS}) \leftrightarrow \\ & [\text{pkra} : \text{PRI\_KEY\_READ\_ACCESS}; \\ & \text{slwa} : \text{SIG\_LOG\_WRITE\_ACCESS}] \end{aligned}$$

In this definition, *USER\_ID*, *SESSION\_ID*, *PRI\_KEY\_READ\_ACCESS*, and *SIG\_LOG\_WRITE\_ACCESS* are the types of user identifiers, session identifiers, and the permissions to access the private keys and the signature log, respectively. *NO\_USER* and *AUTH\_ERRORS* are constants representing error elements. The state of the session manager is formalized by the following  $Z$  schema:

<i>SessionManager</i>
$session\_table : SESSION\_TABLE$ $session\_IDs : \mathbb{F} SESSION\_ID$
$\forall x, y : \text{dom}(session\_table) \bullet$ $(\exists s : SESSION\_ID \bullet s \in \text{dom}(session\_table(x))$ $\wedge s \in \text{dom}(session\_table(y))) \Rightarrow x = y$
$\forall x : \text{dom}(session\_table) \bullet$ $\forall s : \text{dom}(session\_table(x)) \bullet \text{dom}(session\_table(x)) = \{s\}$

In general, a Z schema has a declarative part (above the line) and a predicate part (below the line). The declarative part specifies the schema’s signature as a collection of typed fields, as in a record. The semantics of a Z schema is the set of those records that fulfill the predicate part. In the above schema, the predicate part states that a session identifier is associated with at most one user identifier and, conversely, that each user identifier is associated with at most one session identifier. It follows that each authenticated user has exactly one, unique session identifier.

The DARMA module serves as a communication medium. It can be understood as a record containing shared variables into which both clients and the server “read” and “write” according to their needs. These shared variables record which of the three Windows-side functions are called along with its arguments and the return value from the Linux side. Part of this schema is given below, where we have elided declarations for the arguments and return values for the signature generation and logout functions.

<i>DARMA</i>
$Command : COMMAND$ $User\_authentication\_uid : USER\_ID \setminus \{NO\_USER\}$ $User\_authentication\_pw : \text{seq } CHAR$ $Authentication : SESSION\_ID \setminus SESSION\_ERRORS$ $\vdots$

**Operation Schemas.** Each of the module functions is associated with an operation schema. The association is mostly straightforward. However, one aspect that does require explanation is how we model the input and output to these functions. To do this, we explicitly identify the schema’s local input and output variables (respectively postfixed by “?” and “!”, following the standard Z convention) with their DARMA counterparts and use equality to mimick an assignment.<sup>1</sup> We illustrate this below, for the module functions *AuthenticateUserW* and *AuthenticateUserL*, which were described in Section 2.2.

The schema *AuthenticateUserW* models the identically named function, given in Figure 3. This function is quite simple and essentially acts as a proxy, forwarding values over DARMA. Hence the only thing to model is this communication.

<i>AuthenticateUserW</i>
$userid? : USER\_ID$ $password? : \text{seq } CHAR$ $session\_id! : SESSION\_ID$ $DARMA$
$User\_authentication\_uid = userid?$ $User\_authentication\_pw = password?$ $Command = authenticate\_user$ $session\_id! = Authentication$

<sup>1</sup> Logically, the input and output variables are determined by the DARMA state and could be eliminated. However, not only do they clarify the information flow, they also help to maintain the correspondence between our formal specification and Hitachi’s informal interface descriptions (see Figures 3 and 4) with their explicit inputs and outputs. Note too that, as it is standard for Z, reference to input and output, as well as other imperative notions like assignment, is just a conceptual convenience; the semantics of Z schemas is, of course, the standard declarative one, given by sets of records.

Here the variables  $User\_authenticate\_uid$ ,  $User\_authenticate\_pw$ ,  $Command$ , and  $Authentication$  are state variables from the DARMA state schema. The first two are assigned to the input values  $userid?$  and  $password?$ , modeling user input.  $Command$  represents the name of the function called, named here by the constant  $authenticate\_user$ . Finally the output of the schema,  $session\_id!$ , is assigned  $Authentication$ , representing communication from DARMA (as we will see below, this represents the output of  $AuthenticateUserL$ ).

The actual work in authenticating users and registering session information is carried out on the Linux side by  $AuthenticateUserL$ . Our operation schema here formalizes the description given in Figure 4. Step 1 of the informal description is reflected in the test if the generation of a hash value is successful. Step 2 is modeled in the first *else* branch, which calls the auxiliary function  $AuthenticateUser$  (e.g., with the hash value of the user's password) and which returns either a new session identifier or an error value. The remainder of the specification formalizes how to proceed, depending on whether the hash calculation and authentication succeeded or failed. In the former case ( $Authentication \notin AUTH\_ERRORS$ ), the session manager's state is updated: the session table records, for this user identifier and session identifier, the right to read the user's private key and to update the signature log, and the set of session identifiers is updated with the new session identifier. In the latter case ( $Authentication \in AUTH\_ERRORS$ ), the session manager's state is unchanged. Note that the result of  $AuthenticateUserL$  is stored both in the output  $SessionID!$  and in the DARMA variable  $Authentication$ .

<p style="text-align: center;">— <i>AuthenticateUserL</i> —</p> <p><math>\Delta</math><i>SessionManager</i>  <math>\Xi</math><i>HysteresisSignature</i>  <math>\Xi</math><i>AccessController</i>  <i>username?</i> : <i>USER_ID</i>  <i>password?</i> : seq <i>CHAR</i>  <i>SessionID!</i> : <i>SESSION_ID</i>  DARMA</p> <hr/> <p><i>Command</i> = <i>authenticate_user</i>  <i>Authentication</i> = <b>if</b> <i>hashFailure</i>(<i>User_authentication_pw</i>)    <b>then</b> <i>CRYPT_ERR</i>    <b>else</b> <i>AuthenticateUser</i>(<i>User_authentication_uid</i>,     <i>hash</i>(<i>User_authentication_pw</i>), <i>access_control_list</i>,     <i>session_table</i>, <i>session_IDs</i>)  <i>session_table'</i> = <b>if</b> <i>Authentication</i> <math>\notin</math> <i>AUTH_ERRORS</i>    <b>then</b> <i>session_table</i> <math>\cup</math>     {<i>User_authentication_uid</i> <math>\mapsto</math> {<i>Authentication</i> <math>\mapsto</math>     {<i>pkra</i> == <i>accept_read_prikey</i>, <i>slwa</i> == <i>accept_write_siglog</i> }}}</p> <p>  <b>else</b> <i>session_table</i>  <i>session_IDs'</i> = <b>if</b> <i>Authentication</i> <math>\notin</math> <i>AUTH_ERRORS</i>    <b>then</b> <i>session_IDs</i> <math>\cup</math> {<i>Authentication</i>} <b>else</b> <i>session_IDs</i>  <i>username?</i> = <i>User_authentication_uid</i>  <i>password?</i> = <i>User_authentication_pw</i>  <i>SessionID!</i> = <i>Authentication</i></p>
---

The auxiliary functions used in the above schema are defined using  $Z$ 's axiomatic definitions. For example, *hash* is simply specified as a uninterpreted function over character sequences.

| *hash* : seq *CHAR*  $\rightarrow$  seq *CHAR*

For the proofs that the signature architecture satisfies its requirements, no further properties of *hash* are needed (e.g., that it is a one-way function). Similarly, *hashFailure* is axiomatized as an uninterpreted predicate.

| *hashFailure* :  $\mathbb{P}$ (seq *CHAR*)

In contrast, our definition of  $AuthenticateUser$  specifies its concrete behavior. This function checks the user identifier and the hashed password against an access control list. In the case of a successful authentication,

the function *RegistSessionInformation* is used to generate a fresh session identifier; otherwise the return value is an error element from *AUTH\_ERRORS*.

$$\begin{array}{l}
 \text{AuthenticateUser :} \\
 \quad \text{USER\_ID} \times \text{seq CHAR} \times \text{ACCESS\_CONTROL\_LIST} \times \\
 \quad \text{SESSION\_TABLE} \times \mathbb{F} \text{SESSION\_ID} \\
 \rightarrow \\
 \quad \text{SESSION\_ID} \\
 \hline
 \forall \text{uid : USER\_ID; hpw : seq CHAR; acc\_cnt\_lst : ACCESS\_CONTROL\_LIST;} \\
 \quad \text{ssn\_tbl : SESSION\_TABLE; ssn\_IDs : } \mathbb{F} \text{SESSION\_ID} \bullet \\
 \quad \text{AuthenticateUser(uid, hpw, acc\_cnt\_lst, ssn\_tbl, ssn\_IDs) =} \\
 \quad \text{if uid} \notin \text{dom acc\_cnt\_lst} \text{ then NO\_USER\_ERR} \\
 \quad \text{else if hpw} \neq ((\text{acc\_cnt\_lst(uid)}) \cdot \text{hpwd}) \text{ then INVALID\_PW\_ERR} \\
 \quad \text{else RegistSessionInformation(uid, ssn\_tbl, ssn\_IDs)}
 \end{array}$$

We will refrain from giving further details at this point, as the above should suffice to illustrate the main modeling ideas.

### 3.3. The Process Model

In general, there are many possible ways of enriching a data model with process-oriented aspects, ranging from the use of combined (data/process-oriented) formal methods, e.g., [Fis97, SD97], to working with a fixed notion of abstract machine and execution semantics, e.g., [Abr96]. In our case, we proceed by formalizing the system traces within  $Z$ .

**Architecture as Transition System.** We use  $Z$ 's schema calculus to “wire together” the parts of our data model into an architectural description by specifying how the Windows-side operations interact with the Linux-side operations over DARMA. First, we separately collect all the client-side and server-side operations. We use schema disjunction here to model nondeterministic choice. The two resulting transition relations, *ClientOperation* and *ServerOperation*, model a system where the Windows-side and Linux-side functions may be called in any order and with any values, valid or invalid. Afterwards, we use schema conjunction to model the parallel composition of the client-side operations with the server-side operations and we use existential quantification, again in  $Z$ 's schema calculus, to hide the shared DARMA state.<sup>2</sup> This models synchronous internal communication between the sides. (Internal communication within each side is not modeled here.) The resulting architectural description defines a global transition relation.

$$\begin{array}{l}
 \text{ClientOperation} == \\
 \quad \text{AuthenticateUserW} \vee \text{GenerateSignatureW} \vee \text{LogoutW}
 \end{array}$$

$$\begin{array}{l}
 \text{ServerOperation} == \\
 \quad \text{AuthenticateUserL} \vee \text{GenerateSignatureL} \vee \\
 \quad \text{LogoutL} \vee \text{NopOperationL}
 \end{array}$$

$$\text{System} == \exists \text{DARMA} \bullet \text{ClientOperation} \wedge \text{ServerOperation}$$

Note that *NopOperationL* models a “no-op” operation on the Linux side by simply stuttering the Linux-side state. It results when DARMA is called from the client side, but a client-side error occurs and the step is aborted.

Afterwards, we specify the global state of the system by composing the states of the system components using schema conjunction. (Here *HysteresisSignature* formalizes the part of the Linux-side module's state that manages the signature logs, while the *AccessController* maintains a table with the private keys of users.) Similarly, we specify the initial state, given schemas (not shown here) specifying the initial states of the different modules.

<sup>2</sup> Schema operations such as conjunction and disjunction combine the underlying signatures of their operands. In contrast, existential quantification in  $Z$ 's schema calculus hides the signature of the quantified schema.

$$\begin{aligned} GlobalState &== \\ & \quad SessionManager \wedge HysteresisSignature \wedge AccessController \end{aligned}$$

$$\begin{aligned} Init &== \\ & \quad SessionManagerInit \wedge HysteresisSignatureInit \wedge AccessControllerInit \end{aligned}$$

**System Traces.** The schema *System* formalizes a transition relation, whose state variables range over the input/output variables of all operation schemas (e.g., variables like *username?* and *SessionID!* from *AuthenticateUserW*). To reason about the system behavior, what we actually need is a transition relation expressed in terms of just those variables in *GlobalState* (e.g., state variables such as *session\_table* and *session\_IDs* from the state schema *SessionManager*). Hence, to proceed, we project the transition relation *System* to those state variables in *GlobalState* by existentially quantifying over the remaining variables, like input and output variables. This construction can be elegantly formalized using Z's schema comprehension:

$$Next == \{System \bullet (\theta GlobalState, \theta GlobalState')\}.$$

This builds the relation that consists of pairs  $(\theta GlobalState, \theta GlobalState')$ , whose components formalize the variable tuples (so-called *characteristic bindings* in Z) in the pre-state and post-state.

The pair  $(Init, Next)$ , together with the collection of global states, constitutes a Kripke structure and induces a set of *traces* (or *runs*) in a canonical way. A trace is represented by a function that describes how the global state of the system can evolve over time.

$$\begin{aligned} Traces &== \\ & \quad \{f : \mathbb{N} \rightarrow GlobalState \mid f(0) \in Init \wedge (\forall i : \mathbb{N} \bullet (f(i), f(i+1)) \in Next)\} \end{aligned}$$

### 3.4. Formalizing the Security Requirements

The architecture's informal requirements, given in Section 2.3, are phrased in terms of temporal relationships between system *events*. For example, (R1) states that “the signature architecture must authenticate a user before the user generates a hysteresis signature.” This, and the other two requirements, can be formalized as a set of traces that constitutes a safety property over a set of events and we can formalize the correctness of the architecture by stating that each such property holds for every system trace.

To proceed this way, we must first formalize the relevant events. In model checking, it is common to associate events with different states in a transition system, which correspond to execution events like calls to particular functions. Unfortunately, this leaves open the question of where these events are actually generated. Moreover, it is not well suited to a more abstract, declarative approach to modeling where, rather than program points, there are only sequences of program states. Here we will take an alternate, less operational approach. We introduce abstract *event predicates* that characterize the *state changes* associated with events, i.e., they specify the effect of events rather than their cause. An event predicate, therefore, is a (possibly parameterized) relation over pairs of states that characterizes when a relevant state change occurs.

Let us now turn to (R1), our first requirement. (R1) can be formalized in terms of three event predicates: the session table changes due to a user authenticating himself by logging in; the session table changes due to a user logging out; and the signature log changes due to the generation of a hysteresis signature for some user. Below is our axiomatic definition of these predicates.

$$\begin{array}{|l} In, Out, Sign : USER\_ID \rightarrow (GlobalState \leftrightarrow GlobalState) \\ \hline \forall uid : USER\_ID; s1, s2 : GlobalState \bullet \\ \quad (s1, s2) \in In(uid) \\ \quad \Leftrightarrow uid \notin \text{dom}(s1.session\_table) \wedge uid \in \text{dom}(s2.session\_table) \wedge \\ \quad (s1, s2) \in Out(uid) \\ \quad \Leftrightarrow (uid \in \text{dom}(s1.session\_table) \wedge uid \notin \text{dom}(s2.session\_table)) \wedge \\ \quad (s1, s2) \in Sign(uid) \\ \quad \Leftrightarrow ((uid \in \text{dom}(s1.signature\_log) \wedge uid \in \text{dom}(s2.signature\_log) \\ \quad \wedge (s1.signature\_log(uid) \neq s2.signature\_log(uid))) \\ \quad \vee ((uid \notin \text{dom}(s1.signature\_log) \wedge uid \in \text{dom}(s2.signature\_log)))) \end{array}$$

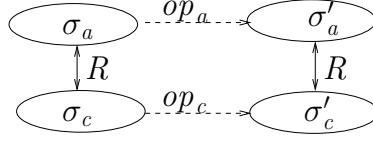


Fig. 5. General Refinement Diagram

We can now directly formalize (R1) in terms of the relative positions (reflecting the relative time) where these predicates hold in the system traces. Our requirement states that at every point where a user changes the signature log, there exists a previous time point where he has logged in, and moreover he has not logged out since then. In other words, there must be a login for the user before the associated signature log entry is changed and his session must still be valid.

$$\begin{aligned} \vdash \forall t : \text{Traces}; n : \mathbb{N}; uid : \text{USER\_ID} \bullet \\ (t(n), t(n+1)) \in \text{Sign}(uid) \\ \Rightarrow (\exists k : 0 \dots (n-1) \bullet (t(k), t(k+1)) \in \text{In}(uid) \\ \wedge (\forall j : (k+1) \dots (n-1) \bullet (t(j), t(j+1)) \notin \text{Out}(uid))) \end{aligned}$$

The other two requirements are formalized similarly.

We conclude with a remark on the close relationship between our use of the event predicates *In*, *Out*, and *Sign* and refinement. These predicates can be understood as operation schemas on a more abstract system description level, which are refined by the concrete operations for logging in, logging out, and signing. For example, *In* can alternatively be formulated by the following operation schema:

$\frac{\text{In2} \quad \Delta \text{GlobalState} \quad uid? : \text{USERID}}{uid? \notin \text{dom}(\text{session\_table}) \wedge uid? \in \text{dom}(\text{session\_table}' )}$
---

These abstract operation schemas can be related to the concrete operation schemas by a refinement relation. Roughly speaking, and as depicted in Figure 5, refinement relates abstract operations  $op_a$  to concrete operations  $op_c$  by relating, under an *abstraction relation*  $R$ , the states of the abstract system  $\sigma_a$  with the states of the concrete system  $\sigma_c$ . The technical details vary depending on which notion of refinement is chosen (e.g., *forward* and *backward simulation* [WD96]), and each impose its own additional conditions, for example, that the domains and ranges of the operations are compatible via  $R$ , etc. For our case-study, it is not difficult to check (and we have done so using Isabelle/HOL) that there is a simple refinement relation where  $R$  is just the bijection on states and proving refinement amounts to showing that the abstract and concrete operations are related one-to-one. For example, the schema *LogoutL* implies *Out* and if two states in a system transition *Next* fulfill *Out*, then *LogoutL* is the only possible operation.

### 3.5. Proofs

All three requirements were proved using the proof environment for HOL-Z. In our comparison in Section 5, we provide statistics on our verification effort. Here we restrict ourselves to a few comments on its overall structure.

The verification required proving 173 theorems. Many of these were simple lemmas, for example, for simplifying expressions, which were then incorporated into Isabelle's automatic proof procedures. The bulk of the preparatory work centered around formalizing and proving (1) properties of operation schemas, (2) architecture decomposition theorems, and (3) global invariants.

With respect to (1), for each operation schema we stated and proved lemmas that characterize its preconditions, postconditions, and invariants in terms of its inputs, outputs, pre-state, and post-state. The theorems proved were of the form

$$OP(in, out, \sigma, \sigma') \Rightarrow COND(in, out, \sigma, \sigma') \Rightarrow \Phi(\sigma, \sigma'),$$

where  $OP$  is an operation schema,  $COND$  a side-condition and  $\Phi$  is one of:

$INV(\sigma, \sigma')$ , expressed in terms of (state variables from) the pre-state  $\sigma$  and the post-state  $\sigma'$ ;

$PRE(\sigma)$ , expressing a condition on the pre-state  $\sigma$ ; or

$POST(\sigma')$ , expressing a condition on the post-state  $\sigma'$ .

An example of such a lemma is the invariant

$$\vdash \text{AuthenticateUserL} \Rightarrow uid : \text{dom}(\text{session\_table}) \Rightarrow \text{session\_table}'(uid) = \text{session\_table}(uid),$$

stating that when a user identifier is in the session table, its entries remain unchanged after another user is authenticated. Note that, as this example illustrates, HOL-Z is syntactically more liberal than Z. This invariant is a HOL-Z formula, but strictly speaking not a Z formula, since it combines Z schema expressions and predicate calculus expressions and it is not closed.

In general, the complexity of proving these lemmas ranged from easy (as in this case) to very high, both in terms of the conceptual work required to understand why they hold and in terms of the proof effort required in Isabelle.

With respect to (2), one of the main lemmas proved was an *architecture decomposition theorem*, which states that the signature architecture can make progress in exactly four ways:

1. an *AuthenticateUserW* step occurs in parallel with an *AuthenticateUserL* step;
2. a *GenerateSignatureW* step starts and aborts due to an internal error while running in parallel with *NopOperationL* (a stuttering step on the Linux side);
3. a *GenerateSignatureW* step occurs in parallel with a *GenerateSignatureL* step; or
4. a *LogoutW* step occurs in parallel with a *LogoutL* step.

By using the Z schema calculus, this theorem can be compactly expressed as:

$$\begin{aligned} \vdash & (\exists \text{DARMA} \bullet \text{AuthenticateUserW} \wedge \text{AuthenticateUserL}) \vee \\ & (\exists \text{DARMA} \bullet \text{GenerateSignatureW} \wedge \text{NopOperationL}) \vee \\ & (\exists \text{DARMA} \bullet \text{GenerateSignatureW} \wedge \text{GenerateSignatureL}) \vee \\ & (\exists \text{DARMA} \bullet \text{LogoutW} \wedge \text{LogoutL}) \\ \Leftrightarrow & \text{System}. \end{aligned}$$

This theorem explains in which ways synchronous communication over DARMA is possible. We use it in the right-to-left direction as a kind of “elimination rule” that decomposes assumptions about steps in traces by case-splitting: if we have a trace  $t$  and a system transition  $(t(n), t(n+1))$ , a property  $P(t(n), t(n+1))$  holds if it holds for the four possible system transitions.

With respect to (3), we proved a large number of global invariants, which are formulas of the form  $\forall t : \text{traces} \bullet INV(t(n), t(n+1))$ . Examples of such invariants are that the signature log monotonically increases and that the domain of the session table and signature log are always bounded by the domain of the table of private keys. These lemmas, as well as the proofs of the three requirements, were proven by induction over the positions in a trace. In the inductive case, the architecture decomposition theorem was applied to decompose the step into possible cases. In each case, either other global invariants or relevant lemmas about properties of operation schemas were used to reason about the consecutive states. Hence, induction and decomposition served as the primary mechanism to reduce the reasoning about global invariants to standard reasoning about local preconditions, postconditions and invariants of operations. In our experience, and perhaps in contrast to common belief, the proofs of these global invariants were not particularly difficult. The main proof effort was spent in establishing local invariants like  $INV(\sigma, \sigma')$ , that is, carrying out conventional reasoning about pre- and post-conditions. We shall return to this observation and provide another perspective on it in Appendix B.

## 4. Modeling and Verification with Spin

The security properties of the signature architecture define authorized sequences of actions, i.e., a property of traces. This suggests building a process-oriented system model from the start that focuses on processes,

relevant aspects of their internal computation, and their communication. We describe such a model in this section and verification by model checking.

### 4.1. Spin

There are a variety of formalisms and tools suitable for process modeling and verification. We have used Spin, one of the most advanced publicly available model checkers, to formalize and check our second model.

Spin is a model checker that supports the design and verification of distributed systems and algorithms. Spin’s modeling language, called PROMELA (PROcess MEtaLanguage), provides a C-like notation for formalizing processes, enriched with process-algebra-like primitives for expressing parallel composition and communication. Properties may be expressed in future-time LTL and Spin implements algorithms for LTL model checking. Appendix A provides a brief introduction to LTL; the PROMELA modeling constructs themselves will be introduced as needed, on-the-fly. For a detailed description of Spin, the reader should consult [Hol04].

### 4.2. Abstraction

As previously noted, one of the main challenges in building a process model is to formalize the system at the right level of abstraction. While this is true for modeling in general, it is particularly crucial when the objective is to arrive at a finite-state system whose properties can be verified by model checking. In the case of the signature architecture, the system’s behavior depends on the data administered, e.g., the values of keys, session identifiers, hash values, and the like, which come from large or even infinite domains. To proceed, we must abstract these data domains into small finite sets, and model functions over data as functions over the corresponding finite sets.

The approach we take is to limit the environment in which the system can be used. In our Z model, the global transition relation *System* modeled interaction with an environment that could call any operation, in any order, and with any value. Whether the transitions represented actions associated with legitimate users, or attackers, was irrelevant, as was the number of such potential users. In our PROMELA model, we will restrict both the number of users that can interact with the signature architecture and the values with which they can call system functions.

In particular, we model the signature architecture as a system operating in an environment comprised of honest users and an attacker. The honest users use the system as intended while the attacker uses the system in perhaps unintended ways and, in particular, attempts to exploit and compromise the system. In both cases, the values with which these principals call system functions are restricted to finite domains. We build the overall system model from submodels that define processes for each of the different subsystems together with the processes that model the normal users and the attacker. We then prove that the desired security properties hold of the system, even in the presence of all the possible malicious actions that can be taken by the attacker. This is analogous to the approach taken in verifying security protocols [Pau98, RSG<sup>+</sup>00, BMV05], where one explicitly models an active attacker who controls the computer network and proves that protocols achieve their properties despite the attacker’s interference.

### 4.3. Modeling Communication

Let’s begin with communication. As suggested by Figures 1 and 2, we can model the signature architecture in terms of five communicating modules. In the PROMELA model, we explicitly model communication; this is in contrast to our Z model (see Section 3.3), where communication was implicitly captured by the way that state transitions are forced to synchronize with DARMA on certain values in the definition of the global transition relation *System*. So here we model each module as a PROMELA process, where each process communicates with other processes over channels. A PROMELA channel is a buffer of some declared (finite) size that holds data of specified types. For each function in a module, we define two channels: one for modeling function calls and the other for modeling the return of computed values. This is depicted in Figure 6, which names the channels used for passing data between processes. All channels are declared to have size zero, which models synchronous communication under PROMELA’s semantics: the process sending

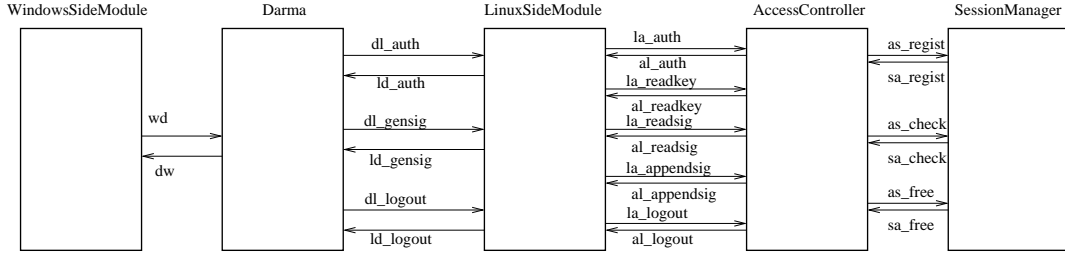


Fig. 6. Modules and Channels

data on a channel and the process receiving data from the channel must rendezvous, i.e., carry out their actions simultaneously.

As the figure shows, between Windows and DARMA we have just one calling channel *wd* and one returning channel *dw*.<sup>3</sup> This reflects that we have only one function in the *Darma* interface. This function is called by marshaling (i.e., packaging) the function arguments together, including the name of the function to be called on the Linux side. We model this by putting all these arguments on the channel. For example, the expression *wd!AuthenticateUser,username,password* (which occurs in our model of a normal user, given shortly), models that the function *AuthenticateUserW* calls *Darma*, instructing *Darma* to call *AuthenticateUserL* with the arguments *username* and *password*.

#### 4.4. Modeling System Users

We now explain our formalization of both normal system users and attackers. The description of the signature architecture in Section 2 describes how the system is intended to be used by normal users. As we will see, it is a simple matter to translate this description into a process that models such users.

The Hitachi documentation describes, in part, the powers and limitations of an attacker; in particular, an attacker cannot access functions on the Linux side. This is a starting point for our formalization of an attacker model, but it leaves many points open, for example, whether an attacker can operate within the Windows-side system as a legitimate user with a valid password, or if he is an outsider, without these abilities. Moreover, it is not specified what the attacker knows, can guess, or can feasibly compute.

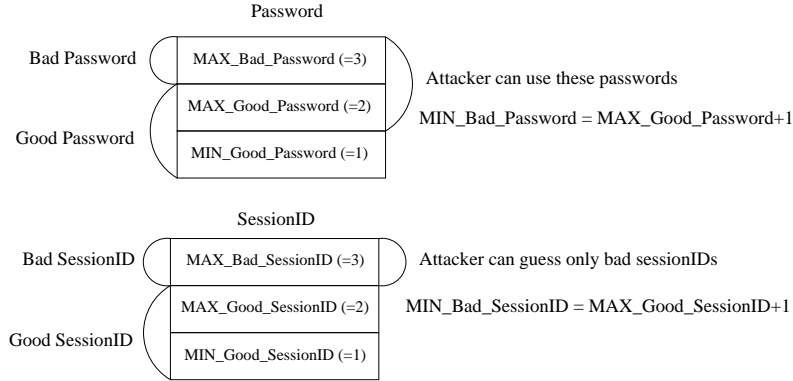
One achieves the strongest security guarantees by proving the safety of a system in the face of the most general and powerful attacker possible. Hence, we model an attacker who cannot only function as a legitimate user of the system, but can also call functions in unintended ways, with arbitrary parameters. Moreover, he knows, or can guess or compute, the names of other users, messages, and message hashes, and of course he knows his own password. However, we assume he can neither guess the passwords nor the session identifiers of other users. (If either of these were possible, then forging signatures would be trivial.)

We summarize these assumptions as follows:

1. The attacker can call *AuthenticateUserW*, *GenerateSignatureW*, and *LogoutW* in any order.
2. The attacker is also a legitimate user with a user name and a password.
3. The attacker knows the names of all users and he can guess messages and message hashes.
4. The attacker can only give his (good) password or a bad guessed password.
5. The attacker cannot guess a good *SessionID*, i.e., one used by other users.
6. Generated *SessionIDs* are always good.

In our PROMELA model, we define sets of objects, namely finite intervals of natural numbers, for modeling the different kinds of objects in the problem domain: names, messages, hash values, and passwords. The key idea is to partition these sets into those things that are known by the attacker (or can be guessed or computed) and those that are not. For example, there is a set of user names, formalized by the set of

<sup>3</sup> Note that we ignore channels for calling the Windows functions since the functions that actually call *AuthenticateUserW*, *GenerateSignatureW*, and *LogoutW* fall outside the scope of our model, that is, we do not consider either the calling context or how the results are used.



**Fig. 7.** Modeling Passwords and Session Identifiers

natural numbers  $\{MIN\_username, \dots, MAX\_username\}$ . We model that the attacker knows, or can guess, any of these names by allowing him to guess (by nondeterministically selecting) any number in this set. However, we partition the ranges corresponding to passwords and session identifiers so that the attacker can only guess “bad” ones, which are ones that are never assigned to normal users. In addition, the attacker also has a “good” password, which allows him to use the system as a normal user and generate a good session identifier. Figure 7 depicts this partitioning, with the concrete values that we later use when model checking. For example, the good passwords are  $\{1, 2\}$ , where 2 represents the attacker’s password. He can only guess passwords in the range  $\{2, 3\}$ , where 3 models a bad password, i.e., one that does not belong to any normal user. As he cannot guess the password 1, he cannot use the system (e.g., to generate a signature) as any user other than himself.

Given this abstraction, it is now a simple matter to model the actions of normal users and the attacker.

**Normal Users.** Figure 8 shows our model<sup>4</sup> of a normal user, which models the steps that such users take when using the signature architecture.

In lines 4 and 5 we model the different possible choices for system users and their messages. The macro  $setrandom(x, lower, upper)$  is defined as:

```
#define setrandom(var, lower, upper)
  atomic{ var = (lower);
    do
      :: break
      :: (var < (upper)) -> var = var + 1
      :: (var == (upper)) -> break
    od;
  skip;
}
```

This routine executes as a single atomic step and uses nondeterministic choice within a loop to set  $x$  to a value,  $lower \leq x \leq upper$ . Hence lines 4 and 5 set the username and password to those of a normal user, chosen nondeterministically from the predefined ranges.

Afterwards, the user generates a hysteresis signature. The lines 8 and 9 model  $AuthenticateUserW$ , which models the equivalent of the identically named  $Z$  schema, which was presented in Section 3.2. Specifically, line 8 models the user calling  $Darma$  on the  $wd$  channel, specifying the execution of the Linux-side user authentication function, along with his username and password. Line 9 models the result returned on the  $dw$  channel: a session identifier (whose value is greater than zero when authentication is successful).

On lines 11–12, a message from the space of possible messages is nondeterministically selected and its message hash is computed. We model  $Hash$  simply as the identity function. Although this does not satisfy the functional requirements of a cryptographic hash function, in particular, that it is a one-way function,

<sup>4</sup> Model excerpts are taken verbatim from our PROMELA model, with the exception of pretty printing, line numbering, and minor simplifications for expository purposes.

```

1 proctype WindowsSideModule_Normal() {
2   byte username, password, sessionID, message, message_hash, signature, result;
3
4   setrandom(username, MIN_Good_Username, MAX_Good_Username);
5   setrandom(password, MIN_Good_Password, MAX_Good_Password);
6
7   do
8     :: wd!AuthUser,username,password;
9     dw?AuthUser,sessionID;
10
11    setrandom(message, MIN_Message, MAX_Message);
12    message_hash = Hash(message);
13
14    wd!GenSig,sessionID,message_hash;
15    dw?GenSig,signature;
16
17    wd!Logout,sessionID,0; /* second argument '0' is dummy */
18    dw?Logout,result
19  od}

```

Fig. 8. User Model

it is adequate for establishing the stipulated properties of our process model, which only rely on passwords and session identifiers being unguessable. On line 14, the user calls *Darma* on the *wd* channel, instructing *Darma* to generate a signature with the session identifier returned from the previous round of authentication and the message hash. The generated signature is returned on line 15. Note that the return value can also indicate an error, for example when the session identifier is invalid.

Lines 17–18 model the user logging out, which invalidates his session identifier.

**The Attacker.** Figure 9 shows the PROMELA process that formalizes our attacker model. Here we see that the attacker can guess an arbitrary user name and message hash (lines 5–6). However, in accordance with the guessing model depicted in Figure 7, he can only guess one good password (*Max.Good.Password*), which allows him to log in as a normal user, or bad passwords (line 7). Similarly, he can only guess bad session identifiers (line 8).

Afterwards, we use a loop with nondeterministic choice to model the attacker repeatedly calling *Darma* (on the *wd* channel) with these guessed values, in any order he likes. Alternatively, as formalized by the last four actions, he can guess new values at any point in time.

This example again illustrates the power of nondeterminism in a process-oriented modeling language. As with the user model, we use it to leave open which values are taken on by variables. This models a system where these variables can take on any value from the specified sets at system runtime. In addition, we use nondeterminism to describe the different possible actions that can be carried out by a user, while allowing the actions to be ordered in any way. This can be contrasted to our Z model which uses relations to formalize a nondeterministic transition relation. In the case of model checking, this nondeterminism typically leads to verification problems with large search spaces whereas in theorem proving it results in the need to perform case splits.

#### 4.5. Modeling Module Functions

The majority of our PROMELA model describes the different functions contained in the system modules. As the Windows-side functions (e.g., *AuthenticateUserW*) have been straightforwardly modeled as calls to DARMA within the user processes, we are left with modeling the Linux-side functions. We model each of them in a standard process-oriented way, by making the control flow of each function explicit, as well as the operations on state variables, and the synchronization with other processes. Functions operating on state variables, which range over finite domains, approximate their counterparts operating over infinite domains, as previously described.

As a representative function, we return to *AuthenticateUserL*, first described in Section 2.2 and specified

```

1 proctype WindowsSideModule_Attacker() {
2   byte username, password, sessionID, signature, dummy, result;
3   bit message_hash;
4
5   setrandom(username, MIN_username, MAX_username);
6   setrandom(message_hash, MIN_Message_Hash, MAX_Message_Hash);
7   setrandom(password, MAX_Good_Password, MAX_Bad_Password);
8   setrandom(sessionID, MIN_Bad_SessionID, MAX_Bad_SessionID);
9
10  do /* Attacker calls these three functions in any order */
11  :: wd!AuthUser,username,password;
12     dw?AuthUser,sessionID
13
14  :: wd!GenSig,sessionID,message_hash;
15     dw?GenSig,signature
16
17  :: wd!Logout,sessionID,dummy;
18     dw?Logout,result
19
20     /* Or, Attacker guesses following values */
21  :: setrandom(username, MIN_username, MAX_username)
22  :: setrandom(message_hash, MIN_Message_Hash, MAX_Message_Hash)
23  :: setrandom(password, MAX_Good_Password, MAX_Bad_Password)
24  :: setrandom(sessionID, MIN_Bad_SessionID, MAX_Bad_SessionID)
25 od}

```

Fig. 9. Attacker Model

```

1  :: dl_auth?username_LINUX,password
2     -> password_hash = Hash(password);
3     if
4     :: (password_hash <= 0) -> sessionID_LINUX = HashFunctionErr;
5         goto DONE_AuthL
6     :: else
7     fi;
8
9     la_auth!username_LINUX,password_hash;
10    al_auth?sessionID_LINUX;
11
12    DONE_AuthL:
13    ld_auth!sessionID_LINUX

```

Fig. 10. AuthenticateUserL

in Z in Section 3.2. Figure 10 shows the part of the PROMELA process that models this function (the module also contains definitions for the other Linux-side functions). This directly models the three steps explained in Section 2.2: calculate a hash value (lines 2–7), authenticate the user (lines 9–10), and return the session identifier (line 13).

#### 4.6. Putting it all together

We build the overall model by composing in parallel the processes defined above. Namely, we compose the two processes formalizing the Windows-side module (as used by normal users and by the attacker) and the processes for the remaining modules. This is depicted in Figure 11. Note that we associate an identifier *lsm* with the process executing the Linux-side module. This will be used during verification to refer to particular labels in an invocation of the *LinuxSideModule* process, as described in the next section.

```

1 init {
2     run WindowsSideModule_Normal();
3     run WindowsSideModule_Attacker();
4     run Darma();
5     lsm = run LinuxSideModule();
6     run AccessController();
7     run SessionManager()}

```

Fig. 11. Initialization Process

## 4.7. Model Checking

We have used the Spin model checker to verify that our model satisfies the security properties listed in Section 2.3. Prior to describing this verification, we first recount how Spin model checking works in general.

Spin can be used to establish that a formula  $\phi$  of future-time linear temporal logic holds of a model  $\mathcal{M}$ , i.e.,  $\mathcal{M} \models \phi$ . The formula  $\phi$  formalizes the desired, or “good”, system behavior and  $\mathcal{M}$  is a Büchi automaton derived from the PROMELA model. For model checking with Spin, one negates  $\phi$ , thereby expressing “bad” system behavior, which in our case is the set of traces that represents security violations. The resulting formula  $\neg \phi$  is converted to a Büchi automaton  $A_{\neg \phi}$ , called a “never claim” in the parlance of Spin, that recognizes precisely this set of bad behaviors. The Spin system then takes the automata  $\mathcal{M}$  and  $A_{\neg \phi}$  as input and reduces model checking to an automata-theoretic problem as described in [VW86] by constructing (on-the-fly) and searching the resulting product automaton. If Spin finds a trace accepted by this automaton, then the trace is a counterexample to  $\phi$  that is accepted by  $\mathcal{M}$  and it explains how the system allows the bad behavior. Alternatively, if Spin succeeds in showing that no such traces exist (by exhaustively showing that there are no acceptance cycles in the product automaton), then it has succeeded in showing  $\mathcal{M} \models \phi$ .

We now illustrate how Spin is used to verify the first requirement described in Section 2.3. This requirement states that the signature architecture must authenticate a user before the user generates a signature. In our Z specification, we formalized this using event predicates that characterized the effect of events. Our PROMELA model is more operational and we can formalize predicates not only in terms of the global system state, but also in terms of the state of the individual processes, e.g., their program counters. This results in direct, albeit lower-level definitions of predicates like *In*, *Out*, and *Out*, which we defined previously.

The following are our PROMELA definitions (written using C-preprocessor notation) of these three predicates.

```

#define in(username,sID)    /* the user username logs in with session identifier sID */
    ((LinuxSideModule[lsm]@DONE_AuthL && username_LINUX == username && sessionID_LINUX == sID)

#define out(sID)            /* the user with session identifier sID successfully logs out */
    (((LinuxSideModule[lsm]@DONE_LogoutL) && (result_LINUX > 0) && (sessionID_LINUX == (sID)))

#define sign(sID)          /* the user with session identifier sID successfully generates signature */
    (((LinuxSideModule[lsm]@DONE_GensigL) && (signature_LINUX > 0) && (sessionID_LINUX == (sID)))

```

In these definitions, we reference labels (using @) in our PROMELA model to formalize that processes have reached certain points in their execution, and we use predicates on variables to express conditions on the system state. For example, consider the predicate *in*, which formalizes the state reached after a user executes the login function. It consists of three conjuncts which formalize that the Linux-side module has: (1) reached the step labeled *Done\_AuthL*, indicating that authentication has completed (see Figure 10); (2) the user being authenticated is *uname*; and (3) the session identifier returned is *sID*. Note that authentication can fail, in which case *sID* is zero. The other predicates are formalized similarly.

Now we can formalize (R1) using these predicates: each signature generated with a session identifier *sID* must be preceded by a login of a user *u*, who is assigned *sID* and has not logged out in the meantime. We express this as

$$\forall s : session. (\exists u : user. (in(u, s)) \text{ before } sign(s)) \wedge \square (out(s) \rightarrow ((\exists u : user. in(u, s)) \text{ before } sign(s))). \quad (1)$$

The first conjunct states that whenever a signature is generated for a session with identifier *s*, it must be preceded by some user *u* who logs in and is allocated *s*. The second states that, after each logout that terminates the session *s*, then again any signature with *s* must be preceded by a login allocating *s*. Here

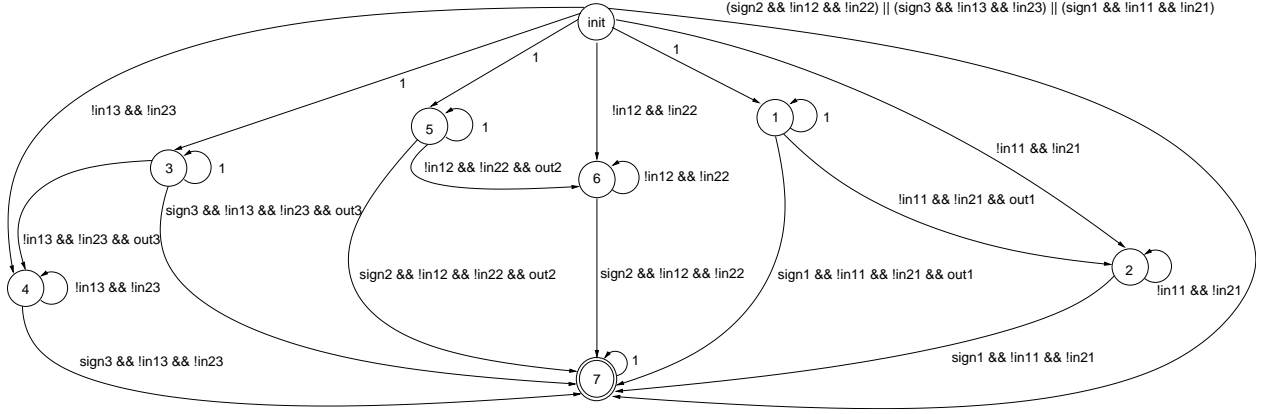


Fig. 12. Büchi Automaton generated for Never Claim for Requirement 1

*user* ranges of the set of all user names and *session* over the set of *valid* session identifiers (i.e., those greater than zero), which result from a successful login. This formula formalizes (R1) as any signature must be preceded by a successful login, and a successful login must also occur after the last logout preceding a signature.

The above is not yet a formula of future-time LTL, to which Spin is limited. To begin with, “before” is not a standard LTL operator (see Appendix A for a definition of LTL). However it can be expressed using the “weak until” modality  $\mathcal{W}$ , where *A before B* is defined as  $(\neg B) \mathcal{W} A$ . Substituting this definition, Equation (1) becomes

$$\forall s : \text{session}. (\neg \text{sign}(s) \mathcal{W} (\exists u : \text{user}. \text{in}(u, s))) \wedge \square (\text{out}(s) \rightarrow (\neg \text{sign}(s) \mathcal{W} (\exists u : \text{user}. \text{in}(u, s))))). \quad (2)$$

Finally, we must eliminate the two quantifiers over sets. Since these sets are finite, we can replace the universally quantified formula by finitely many conjuncts and the existentially quantified formulas by finitely many disjuncts. In particular, the formula  $\forall s : \text{session}. P(s)$  is expanded to  $P(s_1) \wedge P(s_2) \cdots \wedge P(s_n)$ , where  $s_1, \dots, s_n$  are the finitely many model representatives of valid session identifiers and the dual replacement is made in the existential case. So, for example, for a scenario with 3 sessions and 2 users, Equation (2) becomes

$$\begin{aligned} & \neg \text{sign}(s_1) \mathcal{W} (\text{in}(u_1, s_1) \vee \text{in}(u_2, s_1)) \wedge \square (\text{out}(s_1) \rightarrow (\neg \text{sign}(s_1) \mathcal{W} (\text{in}(u_1, s_1) \vee \text{in}(u_2, s_1)))) \wedge \\ & \neg \text{sign}(s_2) \mathcal{W} (\text{in}(u_1, s_2) \vee \text{in}(u_2, s_2)) \wedge \square (\text{out}(s_2) \rightarrow (\neg \text{sign}(s_2) \mathcal{W} (\text{in}(u_1, s_2) \vee \text{in}(u_2, s_2)))) \wedge \\ & \neg \text{sign}(s_3) \mathcal{W} (\text{in}(u_1, s_3) \vee \text{in}(u_2, s_3)) \wedge \square (\text{out}(s_3) \rightarrow (\neg \text{sign}(s_3) \mathcal{W} (\text{in}(u_1, s_3) \vee \text{in}(u_2, s_3))))). \end{aligned} \quad (3)$$

The negation of this formula constitutes our never claim, which we automatically convert into a Büchi automaton with 8 states. For this we use the tool LTL2BA [GO01]. Figure 12 shows the resulting automaton where, to save space, we have abbreviated  $\text{sign}(s_1)$  as *sign1*,  $\text{in}(u_1, s_2)$  as *in12*, etc. Although it would not be easy to specify an automaton like this by hand, it is not too difficult to understand how it works. Roughly speaking, the automaton consists of three parts — right, center, and left — which handle the cases of signatures produced with the first, second, and third session identifier respectively. Consider, for example, the scenario where:

1. the system allows a user to log in with session identifier 1,
2. session 1 is terminated by a log out,
3. some number of steps follow without the session identifier 1 being reassigned by a log in, and
4. a signature is produced in a session with identifier 1.

This scenario, representing a violation of (R1), is accepted by this automaton. Step 1 corresponds to a transition from the initial state to state 1 (a transition labeled 1 can always be taken, including during a login). Actions taken by the system during this session correspond to the self-loop on state 1 (it is immaterial what these actions are, e.g., perhaps a signature is generated or perhaps not). Step 2, the log out, corresponds to the transition from state 1 to state 2. Step 3 corresponds to the self-loop on state 2. Finally step 4 corresponds to the transition from state 2 to the final state 7, an accepting state.

Measurement	HOL-Z	PROMELA/Spin
Model Variants	1	4
Model Size	550 lines	647 lines (average)
Model Bounds	unbounded	2 users, 2 sessions
Property Size	50 lines	53 lines
Proof Size	3662 lines	none
Property Specification Time	2 days	7 days
System Modeling Time	12 days	17 days
Verification Time	19 days	(included above)
Proof Checking Time	12 minutes	96 minutes
Total Time	33 days	24 days
Expert Input Required	60%	15%

**Fig. 13.** Statistics on the Two Verifications

Spin verifies that our model satisfies this property (i.e., no system trace satisfies the never claim) in 45 minutes of computation time.<sup>5</sup> In doing so, it builds a product automaton with over 17 million states and searches almost 60 million transitions. Note that while model checking itself is completely automatic, the model checker may fail to terminate or may exhaust memory. This was often the case in our work and it required considerable interaction to obtain a successful run. We will return to this point in the comparison.

The formalization and verification of the second requirement follows the same pattern. However, the third requirement necessitated a different approach. This requirement involves counting: the signature architecture must generate *only one* hysteresis signature per authentication. While it is straightforward to count up to some finite bound in LTL, it requires using the next-time modality. This means that optimizations, like partial-order reduction [Pel96], which play an important role in making model checking feasible, are no longer sound. As a result, our attempts at model checking this way failed. We took an alternative approach and modified our model, adding an array of counters, one for each session identifier, that tracks the number of signatures generated for each session. We then added a monitor process that runs in parallel and checks that each counter in the array is either set to zero or one. Afterwards we used Spin to automatically verify that this assertion holds for all reachable system states.

## 5. Comparison

In this section, we compare the two different approaches we have taken to modeling and verification. While there have been other general comparisons of theorem proving versus model checking, e.g., [Gup92], and considerable work on their integration, there appear to be few studies that examine their relationship concretely on an in-depth case study. We take up this challenge here and make both quantitative and qualitative comparisons between our two formalizations. The results, we believe, help shed light on the relative strengths and weaknesses of the different approaches.

Note that any such comparison must be made and interpreted with care. The conclusions can differ considerably depending, for example, on the expertise of those carrying out the verification, the specific formalisms and tools used, what is actually measured, and the problem chosen for the comparison. [BK91] contains an in-depth discussion of the difficulties involved here. To ensure an accurate quantitative comparison, we have kept statistics on both verification efforts (the times spent are estimates) and also ensured that each verification was made on an equal footing: Both verifications were carried out by a team consisting of an expert in the formal method and an engineer with limited initial knowledge of the formal method.

Figure 13 summarizes the quantitative differences between the two approaches. We explain these figures below, as well as qualitative differences not captured by these metrics.

<sup>5</sup> All verification times measured in this paper are on a 4 Ghz Intel P4 workstation with 4 GByte RAM running Linux.

## 5.1. Size

In HOL-Z, we built one system model, of 550 lines, against which we verified all three properties. In PROMELA, we built an initial system model, which we adapted afterwards for each of the three properties. The 647 lines of specification is the average size of the four models created. Despite the fact that the HOL-Z model differs substantially from the PROMELA models, they are all of roughly similar size. This stems from the fact that the HOL-Z model is more detailed than the PROMELA models in some respects and more abstract in others. For example, HOL-Z state schemas are more detailed since they define not only data types, but also invariants. On the other hand, HOL-Z operation schemas are typically smaller as they abstractly specify the relationship between states, rather than the sequence of operations used to change states.

For HOL-Z, property size measures the number of lines used to specify the properties and the event predicates used in their definitions. For PROMELA, it is the number of lines used to specify the LTL formulas, the auxiliary predicates, and the changes to the model needed to specify the requirements (including the introduction of a monitor process for the third requirement). The LTL formulas for the first two requirements are quite compact, at least before we expand the set quantifiers.<sup>6</sup> This need not generally be the case: there are temporal requirements whose LTL formalization are non-elementary larger than their formalization in higher-order logic.<sup>7</sup> Finally note that Spin works with the Büchi automata generated from the LTL formulas, and the automata can be exponentially larger. While the automata are generated automatically and it is not actually necessary for users to ever examine them (as we did in Section 4.7), we found that this was helpful in checking that our LTL formulas actually captured the intended requirements.

## 5.2. Time

More time was spent in the theorem-proving approach than in the model-checking approach. The main difference is due to the fact that model checking is automatic as opposed to interactive (the 19 days reflects the time spent interacting with the theorem prover). Folk wisdom is that, because of automation, model checking is much less time consuming than theorem proving. While this is indeed the case for the verification time itself, the *overall* time reduction, about 27%, is not so significant.

However, the numbers point only indirectly to what is probably the most interesting difference: *how* the time was spent. With Spin, once a model and a property are specified, the verification effort is focused on simplifying the problem so that the model checker terminates. In our case, this involved tuning the sizes of the different finite domains as well as introducing abstractions and other simplifications. For example, to reduce verification times, we found it necessary to annotate our model with information (using PROMELA's *atomic* statement) on which sequences of steps can safely be executed atomically, i.e., without interleaving steps from other processes. Moreover, as previously noted, although it is possible to model the requirement (R3) using a never claim, the resulting verification consumed too much computer memory and thus required a different modeling approach. The time spent on these activities (which also includes waiting for runs to fail) was substantial and is reflected both in the increased time taken for system modeling and for property specification.

Note that these efforts are quite different from those required for verification in HOL-Z. Our HOL-Z verification was based on only one model, the general system model. We neither had to work out any abstractions or restrictions in advance nor to make subsequent changes during verification. Hence the specification time was shorter. In return, substantially more time was required for verification. Although some of this time was spent pushing low-level proof details through the Isabelle system, as explained in Section 3.5, much of it concerned discovering, formalizing, and proving auxiliary system invariants, which were required to prove the properties of interest.

Although discovering and proving invariants is a more time-consuming activity than (PROMELA) model

<sup>6</sup> We have counted their size *after* expansion, e.g., counting Equation 3 as 3 lines. Note then that the size of the LTL specification is a function of the size of the finite domains involved and had we been able to model check larger models, our specifications would have also been larger. This is a general problem with specification languages based on propositional logic.

<sup>7</sup> The first-order theory of linear orders  $FO[<]$  with unary predicates, whose formulas can be linearly embedded in higher-order logic, has a non-elementary worst-case complexity. In particular, there are families of formulas whose Büchi automata have non-elementary many states with respect to the formula length.

simplification, it is also a more insightful one. Many of the invariants are interesting in their own right as they lead to a better understanding of why the architecture actually works. Moreover, in our work, they also led to our discovering problems in our original formalization of Hitachi’s requirements. For example, a direct formalization of the first requirement (that signature generation requires a prior login) overlooks the fact that the login session must still be valid, that is, there cannot be a logout between these events. We originally formalized and verified this weaker statement (i.e., the statement that arises from omitting the last conjunct in the theorem statement in Section 3.4 or equivalently the second conjunct in Equation 1) in Spin. Only when working out the invariants in HOL-Z did we realize that the stronger theorem was actually intended and held. Such specification errors can of course be found by careful review, but our experience is that they are much more likely to be made during model checking where users are only confronted with the direct consequences of what they specify.

### 5.3. Expertise needed

In both case studies, expert input was needed, albeit to a different degree and in different places. In both approaches, it was possible for an engineer with limited initial knowledge of the formal method to build the first model after receiving some training for the task. However, for the HOL-Z model, an expert review and restructuring of the model was needed. Finding suitably abstract formulations in Z appears to require more expertise than finding “natural” formulations in PROMELA, which was perceived as a kind of programming language.

In contrast, in the Spin case study, most of the expert help required was in formulating properties. This turned out to be surprisingly tricky. Temporal formula like that of Equation (2) are difficult for novices to formalize, as familiarity with LTL idioms (e.g., specification patterns as in [DAC99]) is helpful for translating statements about the past to those about the future. Also problematic is validating that a given LTL formula actually captures the intended requirements. We did not have these problems with the HOL-Z formalization as the use of the more expressive logic of HOL-Z allowed us to directly quantify over time and make standard comparisons on time points, which is simpler and more intuitive than the use of LTL.

Of course, capturing requirements in LTL does offer certain advantages over their formalization as predicates over traces in HOL. The LTL specifications tend to be more concise than their HOL counterparts. Moreover, in the case of first-order LTL, where verification is no longer decidable, the syntactic form of the temporal formulas may suggest useful proof strategies and inference rules for theorem proving; [CS05, MP91] provide examples of this. Hence an interesting possibility would be to find a middle ground, between these LTL and HOL. One alternative here is to use a more expressive (but still elementary decidable) temporal logic like the real-time extension of LTL proposed in [AH94]. This logic provides constructs for binding variables to time points and making explicit comparisons between time points; this fits well with the kinds of temporal constructs that we used when specifying properties in HOL. Another alternative is to work with a suitable embedding of temporal operators directly within higher-order logic. This allows us to move between these different specification paradigms during specification and theorem proving as well as derive temporal idioms and formally reason about the relationship between different formalizations of requirements. We sketch this option and provide examples in Appendix B.

Finally, the most substantial difference in terms of the expertise required concerned the verification itself. Model checking is push button, but only in theory. In practice, some expert input was required in restructuring and tuning the model so that Spin would terminate. In contrast, for the HOL-Z study considerable hands-on work was required by the expert in order to complete some of the proofs. This is reflected by the 60% expert contribution reported in Table 13.

### 5.4. What was formalized and proved

Finally, the numbers given do not reflect that there were substantial differences, as well as similarities, in what was modeled and verified. Any model of the signature architecture worthy of the name must capture both data-oriented and process-oriented aspects at some level of abstraction. The HOL-Z model does this by providing a data-oriented interface model that specifies the different components of the system state and a relational formalization of the state transition operations. The resulting model is extended to a simple process model with an interleaving trace semantics. This is in contrast to the PROMELA model where

the operational and communication behavior of the interface functions is spelled out concretely, albeit on simplified data domains. SPIN constructs from this a transition system, which has again an interleaving trace semantics. So at first glance we see both similarities (trace semantics) and differences (specification styles and what was actually modeled). We consider below the main differences in more detail as well as their implications.

First, in HOL-Z, we were able to directly model the relevant data domains in their full generality, rather than settling for some finite approximation. This means we did not need to bound, a priori, the size of domains like the set of users, their passwords, session identifiers, and the like. This is in contrast to the PROMELA model, where all data domains were simplified to small finite domains to support model checking, as explained in Section 4.4. The ability to model and reason about infinite domains is a standard benefit of using a rich logic like HOL-Z. Our HOL-Z model is both more general and the theorems proven are significantly stronger.

Second, the two models differ in the way that concurrency and communication are expressed. The PROMELA language is based on the notion of processes which may run in parallel, and communicate either synchronously or asynchronously over communication channels. So direct support for concurrency and communication are built into the language. This is not the case for HOL-Z as the focus is on data modeling, not concurrent process modeling, and hence both concurrency and communication must be explicitly modeled within the language. We introduced concurrency in our HOL-Z model only at the top level of the architecture in fairly coarse-grained client and server steps and we modeled communication by shared schema variables. As the communication model in the architecture was fairly simple, this top-level synchronization was adequate.

One modeling consequence of this is that PROMELA’s bias towards process-oriented modeling led naturally to a model where an attacker is explicitly formalized and whose interface calls are interleaved with those of honest users. This formalization, which has its roots in protocol analysis, has the advantage of explicitly modeling the powers of the attacker for disrupting the system. From the HOL-Z perspective, concurrency is expensive during modeling and even more so during theorem proving. The architecture decomposition theorem is used to make the possible combinations explicit and constructing the resulting proofs would become exorbitantly difficult if fine-grained concurrency were allowed between the components (which is why model checking is the preferred approach in such cases, when it is possible). The HOL-Z model shows another way to proceed: this model makes no constraining assumptions on the clients at all — neither in the order of operations nor in the data sent along DARMA (as long as its types are respected) — and therefore identifies the client with the attacker.

Finally, as noted above, although the two approaches specify the same interface functions, the HOL-Z specification is declarative, while the PROMELA specification is operational. The operational approach, by its very nature, involves commitments to data types and concrete procedures for data manipulation. In contrast, HOL-Z does not require such commitments and this leaves us considerably more flexibility in how the architecture can be refined and for exploring changes. As an example, in the Hitachi architecture, a user may only log in once before logging out again, i.e., a user may be associated with only one session. However, an alternative architecture is one that supports multiple sessions per user. Modeling these kinds of changes in our architecture is trivial. Here, we can specify this alternative simply by deleting the second constraint in the predicate part of the session manager schema (Section 3.2), which requires that each user identifier is associated with at most one session identifier. In this case, almost all of the system invariants proven go through, unchanged.

## 6. Related Work and Conclusions

We have reported on a comparative case study in verifying a security architecture using both theorem proving and model checking. As noted in Section 5, we know of no other comparisons along these lines, although there have been comparisons made of other formal methods and verification approaches. Examples include comparisons between different specification languages [ABL86], between different theorem provers [BK91], and between different automated analysis techniques (including model checking, static analysis, and testing) [ACD<sup>+</sup>99, Cor96, BDG<sup>+</sup>04]. The domain we have chosen, architecture modeling and verification, is one that has been extensively studied before using both data-oriented and process-oriented formal methods, e.g. [AAG95, CAB<sup>+</sup>98, JS00, SG96, WVF97]. However, the focus of these other studies has been different, namely showing the suitability of the different individual formal methods for the verification task.

Our work also sheds light on the suitability of using Z and the associated HOL-Z environment for formalizing and verifying architectures that combine data-oriented and process-oriented aspects. It should also be noted in this regard that our modeling and verification of the signature architecture is one of the largest case studies made using HOL-Z. Previous case studies also include a security architecture (for controlling access to a repository) [BW03], but there the emphasis was on data refinement, rather than the verification of temporal properties of system runs. The studies are complementary in that together they illustrate how HOL-Z can be used to formalize, verify, and refine architectures at different levels of abstractions, covering both data-oriented and process-oriented aspects.

There are a number of avenues open for future work. To begin with, the theorems we proved by model checking are weaker than those proved using HOL-Z. In some situations however it may be possible to verify the correctness of the abstractions used, i.e., that the verification of the small finite model used entails the verification of the corresponding infinite state system whose state variables range over infinite data domains. Techniques based on data independence, such as those of [Low98, RB99], may help automate this task.

Another direction concerns the way we modeled cryptographic functions. In the HOL-Z model, cryptographic functions for hashing and signing were simply treated as uninterpreted function symbols, specified just by their type. This leaves open all possible implementations, faithful or not to the standard cryptographic requirements for such functions. In the PROMELA model, we had to take the other extreme and commit to concrete computable functions. Our formalization has the property that certain actions, like guessing values, are impossible as opposed to being highly improbable, which is the case in actual cryptography. As the properties we examined were possibilistic rather than probabilistic, this was not a problem. However, it would be interesting to investigate whether cryptographically sound abstractions such as those of [Can01, BPW03] could be usefully employed in this setting.

Finally, while we learned much ourselves from this comparative case study, and hope that others can also profit from it, it is of course only one data point documenting the issues and tradeoffs involved. Our case study combined both data-oriented and process-oriented aspects, but most of the complexity was in the data-modeling side and, as noted in Section 5.4, the HOL-Z communication model was coarse grained. For this combination, there were striking benefits from using the HOL-Z approach. But for a system with simpler operations and more complex process interaction, the conclusions might be quite different. An example of this would be architectures whose complexity is dominated by the communication protocols employed: here we would expect model checking to have the upper-hand. Said another way, the tradeoff are not absolute, but relative to the problem under consideration. Additional comparative case studies could contribute to our understanding of this relationship and to the development of refined guidelines for the use of different formal methods.

## Acknowledgments

We thank Jean-Raymond Abrial, Achim Brucker, Christoph Sprenger, Felix Klaedtke, and Ernst-Rüdiger Olderog for helpful discussions on the contents of this paper.

## References

- [AAG95] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(4):319–364, October 1995.
- [ABL86] J.-R. Abrial, E. Börger, and H. Langmaack. *Formal methods for industrial applications: Specifying and programming the steam boiler control*, volume 1165 of *LNCS*. Springer-Verlag, 1986.
- [Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.
- [ACD<sup>+</sup>99] G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Pasareanu, and S. F. Siegel. Comparing finite-state verification techniques for concurrent software. Technical report, Amherst, MA, USA, 1999.
- [AH94] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–203, 1994.
- [ASS<sup>+</sup>99] Toshiaki Arai, Tomoki Sekiguchi, Masahide Satoh, Taro Inoue, Tomoaki Nakamura, and Hideki Iwao. Darma: Using different OSs concurrently based on nano-kernel technology. In *Proc. 59th-Annual Convention of Information Processing Society of Japan*, volume 1, pages 139–140. Information Processing Society of Japan, 1999. In Japanese.
- [BDG<sup>+</sup>04] Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2-3):167–198, 2004.
- [BK91] David Basin and Matt Kaufmann. The Boyer-Moore Prover and Nuprl: An experimental comparison. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 90 – 119. Cambridge University Press, 1991.

- [BKTW04] David Basin, Hironobu Kuruma, Kazuo Takaragi, and Burkhart Wolff. Specifying and verifying hysteresis signature system with HOL-Z. Technical Report 471, ETH Zürich, January 2004. Available at the URL <http://kisogawa.inf.ethz.ch/WebBIB/publications/papers/2005/HSD.pdf>.
- [BMV05] David Basin, Sebastian Mödersheim, and Luca Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, June 2005. Published online December 2004.
- [BPW03] Michael Backes, Birgit Pfützmann, and Michael Waidner. A composable cryptographic library with nested operations. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 220–230, New York, NY, USA, 2003. ACM Press.
- [BRW03] Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, February 2003.
- [BW03] Achim D. Brucker and Burkhart Wolff. A case study of a formalized security architecture. In *Electronic Notes in Theoretical Computer Science*, volume 80. Elsevier Science Publishers, 2003.
- [CAB<sup>+</sup>98] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.
- [Can01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS '01: Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science*, page 136. IEEE Computer Society, 2001.
- [Cor96] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–180, 1996.
- [CS05] Claudio Castellini and Alan Smail. Proof planning for first-order temporal logic. In *Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2005.
- [DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 411–420. IEEE Computer Society Press, 1999.
- [Fis97] Clemens Fischer. CSP-OZ: A combination of Object-Z and CSP. In *Proceedings of FMOODS'97: Formal Methods for Open Object-Based Distributed Systems*, volume 2, pages 423–438. Chapman & Hall, 1997.
- [GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)*, number 2102 in *Lecture Notes in Computer Science*, pages 53–65. Springer Verlag, 2001.
- [Gup92] Aarti Gupta. Formal hardware verification methods: A survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992.
- [Hol04] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, 2004.
- [Int] International Standard ISO/IEC 13568:2002. Information technology — Z formal specification notation — syntax, type system and semantics.
- [JS00] Daniel Jackson and Kevin Sullivan. COM revisited: tool-assisted modelling of an architectural framework. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 149–158. ACM Press, 2000.
- [Low98] Gavin Lowe. Towards a completeness result for model checking of security protocols. In *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*, pages 96–105. IEEE Computer Society Press, 1998.
- [MP91] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theoretical Computer Science Journal*, 83(1):97–130, 1991.
- [MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
- [NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL — A Proof Assistant for Higher-Order Logic, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
- [Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
- [Pel96] Doron Peled. Combining partial order reductions with on-the-fly model checking. *Formal Methods in System Design*, 8:39–64, 1996.
- [Pro05] Promela proofs scripts for signature system case study. URL <http://people.inf.ethz.ch/basin/spin-models.tar>, 2005.
- [RB99] A. W. Roscoe and Philippa J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(1):147–190, 1999.
- [RSG<sup>+</sup>00] Peter Y. A. Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and A. W. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2000.
- [SD97] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In *Proceedings of the International Conference of Formal Engineering Methods*, pages 293–302. IEEE Computer Society Press, 1997.
- [SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall Publishing, 1996.
- [SM02] Seiichi Susaki and Tsutomu Matsumoto. Alibi establishment for electronic signatures. *Information Processing Society of Japan*, 43(8):2381–2393, 2002. In Japanese.
- [Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, New Jersey, second edition, 1992.
- [VW86] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.
- [WD96] Jim Woodcock and Jim Davies. *Using Z*. Prentice-Hall International, New Jersey, 1996.
- [WVF97] Jeannette Wing and Mandana Vaziri-Farahani. A case study in model checking software systems. *Science of Computer Programming*, 28:273–299, 1997.

## A. Linear Temporal Logic

**Syntax.** Let a set of *assertions* (also called *state formulas*) be given. A (*full*) *LTL formula* is built from assertions using the Boolean connectives  $\neg$  and  $\wedge$  and the temporal operators  $\bigcirc$  (Next),  $\ominus$  (Previous),  $\mathcal{U}$  (Until), and  $\mathcal{S}$  (Since). Future-time LTL is the subset of full LTL where the only temporal operators are  $\bigcirc$  and  $\mathcal{U}$ . Note that we will also employ additional propositional constants (**T** and **F**) and connectives (defined as standard) as well as the following defined temporal operators:

$$\begin{aligned} \diamond p &= \mathbf{T} \mathcal{U} p && \text{(sometime, eventually)} \\ \square p &= \neg \diamond \neg p && \text{(always, henceforth)} \\ p \mathcal{W} q &= \square p \vee (p \mathcal{U} q) && \text{(unless, weak until)} \end{aligned}$$

**Semantics.** Let a set of states  $\sigma$  be given. Semantically, each assertion defines a predicate over states of type  $\sigma \rightarrow \text{bool}$ . A *model* is an infinite trace  $t$  of type  $\text{nat} \rightarrow \sigma$ . Given a model  $t$ , we define the *satisfaction* of a formula  $p$  at position  $j \geq 0$ , written as  $(t, j) \models p$ , by cases:

$$\begin{aligned} (t, j) \models p &\Leftrightarrow p(t(j)) && \text{(assertion)} \\ (t, j) \models \neg p &\Leftrightarrow \neg((t, j) \models p) && \text{(negation)} \\ (t, j) \models p \wedge q &\Leftrightarrow (t, j) \models p \wedge (t, j) \models q && \text{(conjunction)} \\ (t, j) \models \bigcirc p &\Leftrightarrow (t, j+1) \models p && \text{(next)} \\ (t, j) \models p \mathcal{U} q &\Leftrightarrow \exists k \geq j. (t, k) \models q \wedge (\forall i. j \leq i < k \Rightarrow (t, i) \models p) && \text{(until)} \\ (t, j) \models \ominus p &\Leftrightarrow j > 0 \wedge (t, j-1) \models p && \text{(previous)} \\ (t, j) \models p \mathcal{S} q &\Leftrightarrow \exists k \leq j. (t, k) \models q \wedge (\forall i. k \leq i < j \Rightarrow (t, i) \models p) && \text{(since)} \end{aligned}$$

Given a Kripke structure  $K$ , we write  $K \models p$  iff  $(t, 0) \models p$  for all  $t$  of  $K$ .

## B. HOL requirements in an LTL style

In Section 5.3 we observed that LTL has advantages over HOL in terms of conciseness and inference rules. We now return to this point and show that it is possible to have the best of both logics via an embedding of LTL within HOL. Note that all of the definitions and equivalence proofs given here have been checked in Isabelle/HOL.

Recall from Section 3.4 that event predicates constitute abstract actions. Given a Kripke structure  $K = (\text{Init}, \text{Next})$ , we can transform it into a Kripke structure suitable for reasoning about actions, a so-called *action Kripke structure*, by the bijection  $K_A$  of type  $(\mathbb{P}\sigma \times (\sigma \leftrightarrow \sigma)) \rightarrow (\mathbb{P}(\sigma \times \sigma) \times ((\sigma \times \sigma) \leftrightarrow (\sigma \times \sigma)))$  defined by:

$$K_A(\text{init}, \text{trans}) \equiv (\{(s, s') \mid s \in \text{init} \wedge (s, s') \in \text{trans}\}, \{(s, s'), (t, t') \mid s' = t \wedge (s, s') \in \text{trans} \wedge (t, t') \in \text{trans}\})$$

In contrast to  $K$ , the assertions of  $K_A(\text{Init}, \text{Next})$  are predicates over *pairs of states*, i.e., actions from our HOL-Z model.

It is now a simple matter to translate the HOL formalizations of our three requirements into LTL, which we illustrate here with (R1). One such formulation is

$$K_A(\text{Init}, \text{Trans}) \models \square (\text{Sign}(\text{uid}) \longrightarrow (\neg \text{Out}(\text{uid}) \mathcal{S} \text{In}(\text{uid}))), \quad (4)$$

which says that “every time the user  $\text{uid}$  produces a signature, the user has previously logged in, and not logged out since then”. It is not difficult to prove that this is equivalent to our HOL formulation given in Section 3.4. Unfolding this definition yields

$$\begin{aligned} \forall t \in \text{traces}(K_A(\text{Init}, \text{Trans})). \\ \forall k. (t k, t(k+1)) \in \text{Sign}(\text{uid}) \longrightarrow \\ (\exists ka \in \{0..k\}. \\ (t ka, t(ka+1)) \in \text{In}(\text{uid}) \wedge \\ (\forall j \in \{ka+1..k\}. (t j, t(j+1)) \notin \text{Out}(\text{uid}))). \end{aligned}$$

Although this is not identical to our original formulation, it is equivalent given that the actions  $\text{Sign}$ ,  $\text{In}$  and

$$\begin{array}{l}
\forall t.t0 \notin P \wedge INV(t0) \quad (\text{Init}) \\
\forall t n.INV(tn) \wedge tn \notin R \longrightarrow INV(r(n+1)) \quad (\text{Inv}) \\
\forall t n.tn \notin Q \longrightarrow INV(t(n+1)) \quad (\text{InvEntry}) \\
\forall t n.tn \in R \longrightarrow \neg INV(t(n+1)) \quad (\text{InvExit}) \\
\forall x.INV x \longrightarrow x \notin P \quad (\text{InvConcl}) \\
\hline
K \models \Box(P \rightarrow \ominus(Q \mathcal{S} R))
\end{array}$$

Fig. 14. Rule SafeSince

*Out* are disjoint. Moreover, using standard equivalences (see [MP92]), it is possible to convert this formula into a future-time LTL formula. In particular, using

$$\Box(P \longrightarrow (Q \mathcal{S} R)) \Leftrightarrow \neg P \mathcal{W} R \wedge \Box(\neg Q \longrightarrow \neg P \mathcal{W} R)$$

we can rewrite and simplify (4) to

$$K_A(\text{Init}, \text{Trans}) \models \neg \text{Sign}(uid) \mathcal{W} In(uid) \wedge \Box(\text{Out}(uid) \longrightarrow \neg \text{Sign}(uid) \mathcal{W} In(uid)).$$

At this point, the connection to the LTL formalization of (R1) in the Spin case study (see Section 4.7, Equation (2)) becomes apparent: both have the identical temporal structure. The only differences concern the way the particular HOL-Z actions (versus PROMELA assertions) and associated information (user identifiers and session identifiers) are formalized.

The above shows one use of HOL specifications in a temporal style: we can relate higher-order logic specifications to those in weaker logics like LTL. We conclude here by describing a second use: the LTL embedding can serve as the basis for deriving useful proof rules. We given an example of this, which also sheds some light on our observation in Section 3.5 that most of our effort in HOL-Z verification was devoted to local reasoning about preconditions and postconditions of operations.

An alternative, equivalent, formalization of (R1) is

$$K_A(\text{Init}, \text{Trans}) \models \Box \text{Sign}(uid) \longrightarrow \ominus(\neg \text{Out}(uid) \mathcal{S} In(uid)). \quad (5)$$

Figure 14 presents a derived rule for reasoning about specifications of this form, motivated by the inference rule SAFE of [MP91, page 9]. We have used this rule to reason about temporal formalizations of the three requirements (R1) – (R3).<sup>8</sup> In this rule the  $t$  range over traces and the  $n$  range over natural numbers. Note that when interpreted over the traces of an action Kripke structure,  $tn$  refers to a pair of states  $(tn, t(n+1))$ .

When applied, the premises of the rule often suggest an appropriate invariant  $INV$ . For example, when applied to verify (5),  $P$  is instantiated by  $\text{Sign}(uid)$ ,  $Q$  by  $\neg \text{Out}(uid)$ , and  $R$  by  $In(uid)$ . From the structure of the premises we can easily guess the invariant  $INV$ : the user  $uid$  should not be authenticated at the beginning of the state transition. This invariant can be easily expressed in the terms of the system model:  $uid$  must not be in the domain of the initial session table. Applying SafeSince for this instance reduces the overall burden to prove global (R1) into the following proof obligations, all of which are local:

1. at the beginning, the generation of a signature is not possible and no user is authenticated (**Init**),
2. if a user is not authenticated and no login occurs, he remains unauthenticated (**Inv**),
3. if a logout occurs, i.e.  $\neg \neg \text{Out}(uid)$ , the user should not be authenticated (**InvEntry**),
4. if a login occurs, the user should be authenticated (**InvExit**), and
5. if the user is not authenticated, no generation of a signature is possible (**InvConcl**).

The proof of SafeSince itself is a routine induction over the position in a trace. In contrast, the proof of (**InvConcl**) is two orders of magnitudes larger and substantially more complex. Discharging the other proof obligations is easy since they only use abstract system operations themselves; thus, they profit from the abstraction of the underlying refinement.

<sup>8</sup> In the case study we verified non-temporal, higher-order formalizations, as we reported on in Section 3.4. Afterwards completion of the case study, in order to better understand the relationships between the two models, we carried out the temporal formalization in HOL and experiments using derived (temporal) proof rules. The statistics reported on in Section 5 are for the original case study, and do not include the post-hoc reformulation and experiments.