Verification of a Signature Architecture with HOL-Z

David Basin[1], Hironobu Kuruma[2], Kazuo Takaragi[2], and Burkhart Wolff[1]

[1] ETH Zurich, CH-8092 Zurich, Switzerland
{basin,bwolff}@inf.ethz.ch
[2] Hitachi Systems Development Laboratory, Yokohama Japan
{kuruma,takara}@sdl.hitachi.co.jp

Abstract. We report on a case study in using HOL-Z, an embedding of Z in higher-order logic, to specify and verify a security architecture for administering digital signatures. We have used HOL-Z to formalize and combine both data-oriented and process-oriented architectural views. Afterwards, we formalized temporal requirements in Z and carried out verification in higher-order logic.

The same architecture has been previously verified using the SPIN model checker. Based on this, 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 this approach and provides evidence that, in the hands of experienced users, theorem proving is neither substantially more time-consuming nor more complex than model checking.

1 Introduction

While there is increasing consensus about the usefulness of formal methods for developing and validating 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 the system they emphasize, the proof techniques used, etc. When most of the complexity of the system 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 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 sequencing? Here

J.S. Fitzgerald, I.J. Hayes, and A. Tarlecki (Eds.): FM 2005, LNCS 3582, pp. 269–285, 2005.

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) [2], 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 the processes 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 can change.

Here we present a model of the signature architecture that combines data-oriented and process-oriented aspects. We describe the system's state and its state transitions in the specification language Z [14]. As Z is a very rich specification language, we also use it to formalize a simple process model describing the system's semantics in terms of the set of its traces, i.e., those state sequences possible. This provides a basis for naturally formalizing the system's integrity requirements as trace requirements and carrying out verification by induction over the set of traces. Our first contribution in this paper is to show how the use of a sufficiently expressive data-modeling language provides a foundation for formalizing a trace-based model of process interaction. Thus, 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. Moreover, via the embedding of Z in higher-order logic (HOL-Z), we can prove system correctness by theorem proving within the Isabelle/HOL system [6, 12].

In a previous case study [5], the same architecture was formalized and verified using the SPIN model checker [10]. Our second contribution is to 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). Perhaps surprisingly, our experience shows 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.

Overall, our modeling and verification of 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) [7], 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 and process-oriented aspects.

Organization. In Section 2, we provide an informal overview of both the signature architecture and its security requirements. We describe our formalization of the architecture in Section 3 and its properties and correctness proofs in Section 4. In Section 5, we conclude with an in-depth comparison with a previous case study based on model checking. Note that all definitions and complete proof scripts for this case study are given in [4].

2 The Signature Architecture

2.1 Overview

The signature architecture is based on two ideas. The first is that of a *hysteresis signature* [15], 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 [2] 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 user operating system space. We call this the "Windows-side module" to reflect the

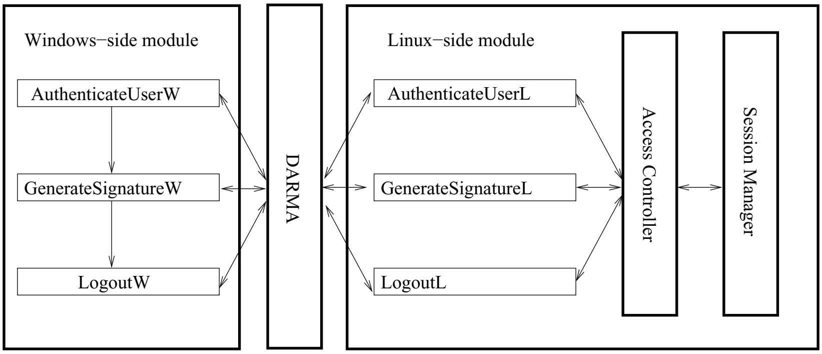

Fig. 1. The Signature Architecture

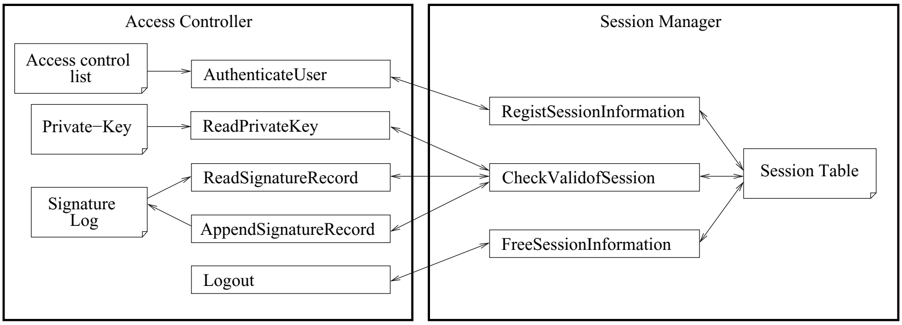

Fig. 2. The Access Controller and Session Manger Modules

(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.
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

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*

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 *CrypotErr* (≤ 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*

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.

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.

3 Formal Model

3.1 Formal Method Used

For our work, we have used Z as our modeling language and the environment Isabelle/HOL-Z for theorem proving. As Z is well established and extensively documented, e.g., [11, 14, 16], we will assume the reader's familiarity with it. HOL-Z [6] is a system built upon Isabelle/HOL [12]. It provides a front end for creating "literate specifications", where specifications are mixed with informal explanations and are constructed as 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 system state and operations is basically standard and closely follows Hitachi's informal specification: we formalize a state schema for each of the different modules and an operation schema for each function.

State Schemas. As examples, we present two state schemas: the session manager and DARMA. The session manager maintains a session table, which associates user names and session identifiers to information on access permissions for keys and the signature log.

$$\begin{array}{l}
SESSION\_TABLE == \\
\quad (USER\_ID \setminus \{NO\_USER\}) \nrightarrow \\
\qquad (SESSION\_ID \setminus AUTH\_ERRORS) \nrightarrow \\
\qquad\quad [pkra : PRI\_KEY\_READ\_ACCESS\ ; \\
\qquad\quad slwa : SIG\_LOG\_WRITE\_ACCESS]
\end{array}$$

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 permissions on private keys and signature log access, respectively. NO_USER and $AUTH\_ERRORS$ are constants representing error elements. The session manager also stores the set of session identifiers currently in use.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SessionManager}} \\
session\_table : SESSION\_TABLE \\
session\_IDs : \mathbb{F}\, SESSION\_ID \\
\hline
\quad \forall x, y : \mathsf{dom}(session\_table) \bullet \\
\qquad\qquad (\exists s : SESSION\_ID \bullet s \in \mathsf{dom}(session\_table(x)) \\
\qquad\qquad\qquad \land\ s \in \mathsf{dom}(session\_table(y))) \Rightarrow\ x = y \\
\quad \forall x : \mathsf{dom}(session\_table) \bullet \\
\qquad \forall s : \mathsf{dom}(session\_table(x)) \bullet \mathsf{dom}(session\_table(x)) = \{s\} \\
\hline
\end{array}
$$

The predicate part of this schema 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. From this predicate, it follows that each authenticated user has exactly one, unique session identifier.

The DARMA module serves as a communication medium. Its state records 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.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DARMA}} \\
Command : COMMAND \\
User\_authentication\_uid : USER\_ID \setminus \{NO\_USER\} \\
User\_authentication\_pw : \mathrm{seq}\ CHAR \\
Authentication : SESSION\_ID \setminus \{x : SESSION\_ERROR \bullet Inr(x)\} \\
\qquad \vdots \\
\hline
\end{array}
$$

Operation Schemas. Each of the module functions is associated with an operation schema. The association is mostly straightforward, although one aspect that requires explanation is the way that we model DARMA's use as a communication medium. To formalize this, each operation schema includes a copy of the DARMA state and explicitly relates the schema's local input/output variables (respectively postfixed by "?" and "!", following the standard Z convention) with their DARMA counterparts.[1] 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.

$AuthenticateUserW$

$userid? : USER\_ID$
$password? : \operatorname{seq} CHAR$
$session\_id! : SESSION\_ID$
$DARMA$

$User\_authentication\_uid = userid?$
$User\_authentication\_pw = password?$
$Command = authenticate\_user$
$session\_id! = Authentication$

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 the input values *userid*? and *password*?, coming from the user. *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 of the hash value. Step 2 is modeled in the first *else* branch, using an auxiliary function for user authentication, 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*.

[1] 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 bindings.

$$\begin{array}{|l}
\multicolumn{1}{l}{\text{--- } \mathit{AuthenticateUserL} \text{ ---}} \\
\Delta \mathit{SessionManager} \\
\Xi \mathit{HysteresisSignature} \\
\Xi \mathit{AccessController} \\
\mathit{username}? : \mathit{USER\_ID} \\
\mathit{password}? : \mathrm{seq}\, \mathit{CHAR} \\
\mathit{SessionID}! : \mathit{SESSION\_ID} \\
\mathit{DARMA} \\
\hline
\mathit{Command} = \mathit{authenticate\_user} \\
\mathit{Authentication} = \mathbf{if}\ \mathit{hashFailure}(\mathit{User\_authentication\_pw}) \\
\quad \mathbf{then}\, \mathit{CRYPT\_ERR} \\
\quad \mathbf{else}\, \mathit{AuthenticateUser}(\mathit{User\_authentication\_uid}, \\
\qquad \mathit{hash}(\mathit{User\_authentication\_pw}), \mathit{access\_control\_list}, \\
\qquad \mathit{session\_table}, \mathit{session\_IDs}) \\
\mathit{session\_table}' = \mathbf{if}\ \mathit{Authentication} \notin \mathit{AUTH\_ERRORS} \\
\quad \mathbf{then}\, \mathit{session\_table} \cup \\
\qquad \{\mathit{User\_authentication\_uid} \mapsto \{\mathit{Authentication} \mapsto \\
\qquad\quad \langle\!| \mathit{pkra} == \mathit{accept\_read\_prikey}, \mathit{slwa} == \mathit{accept\_write\_siglog} |\!\rangle\}\} \\
\quad \mathbf{else}\, \mathit{session\_table} \\
\mathit{session\_IDs}' = \mathbf{if}\ \mathit{Authentication} \notin \mathit{AUTH\_ERRORS} \\
\quad \mathbf{then}\, \mathit{session\_IDs} \cup \{\mathit{Authentication}\}\ \mathbf{else}\, \mathit{session\_IDs} \\
\mathit{username}? = \mathit{User\_authentication\_uid} \\
\mathit{password}? = \mathit{User\_authentication\_pw} \\
\mathit{SessionID}! = \mathit{Authentication} \\
\hline
\end{array}$$

In this schema, the functions *hashFailure* and *AuthenticateUser* are defined separately by axiomatic definitions. For example, *AuthenticateUser* checks the user identifier and the hashed password against an access control list. In the case of a successful authentication, a new session identifier is generated.

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., [8, 13], to working with a fixed notion of abstract machine and execution semantics, e.g., [1]. 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: This transition relation models a system where the Windows-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. 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.

$$\begin{aligned} &ClientOperation == \\ &\quad AuthenticateUserW \lor GenerateSignatureW \lor LogoutW \end{aligned}$$

$$\begin{aligned} &ServerOperation == \\ &\quad AuthenticateUserL \lor GenerateSignatureL \lor \\ &\quad LogoutL \lor NopOperationL \end{aligned}$$

$$System == \exists DARMA \bullet ClientOperation \land ServerOperation$$

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 (*HysteresisSignature* formalizes the part of the Linux-side module's state that manages the signature logs). Similarly, we specify the initial state, given schemas (not shown here) specifying the initial states of the different modules.

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

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

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. 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.

Afterwards, we define the set of traces. Each trace is represented by a function that describes how the global state of the system can evolve over time.

$$
\begin{array}{l}
Traces == \\
\quad \{f : \mathbb{N} \to GlobalState \mid f(0) \in Init \land (\forall i : \mathbb{N} \bullet (f(i), f(i+1)) \in Next)\}
\end{array}
$$

4 Properties and Proofs

4.1 Formalizing the Security Requirements

The architecture's informal requirements, given in Section 2.3, are phrased in terms of temporal relationships between *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.

First we must 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 there are no program points, 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. The formalizations of the other two requirements are similar. (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 a generated hysteresis signature), for some user. Below is an axiomatic definition formalizing the first of these predicates.

$$
\begin{array}{|l}
userDoesLogin : USER\_ID \to (GlobalState \leftrightarrow GlobalState) \\
\hline
\forall uid : USER\_ID \ ; s1, s2 : GlobalState \bullet \\
\quad (s1, s2) \in userDoesLogin(uid) \\
\iff \\
\quad uid \notin \mathsf{dom}(s1.session\_table) \land uid \in \mathsf{dom}(s2.session\_table)
\end{array}
$$

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 the user 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.

$$\begin{array}{l} \vdash \ \forall t : Traces\ ; n : \mathbb{N}\ ; uid : USER\_ID \bullet \\ \qquad\qquad (t(n), t(n+1)) \in siglogChanges(uid) \\ \qquad\quad \Rightarrow \\ \qquad\qquad (\exists k : 0 \mathinner{..} (n-1) \bullet (t(k), t(k+1)) \in userDoesLogin(uid) \\ \qquad\qquad\quad \wedge (\forall j : (k+1) \mathinner{..} (n-1) \bullet \\ \qquad\qquad\qquad (t(j), t(j+1)) \notin userDoesLogout(uid))) \end{array}$$

Note that we have formalized our requirement in terms of consecutive pairs of time points and relationships between time points. An alternative, also possible in Z, would be to embed the operators of a temporal logic like LTL over our traces in order to express these dependencies using temporal modalities.

4.2 Proofs

All three requirements were proved using the proof environment for HOL-Z. 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 proven 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 Φ is one of:

$INV(\sigma, \sigma')$, expressed in terms of (state variables from) the pre-state σ and the post-state σ';
$PRE(\sigma)$, expressing a condition on the pre-state σ; or
$POST(\sigma')$, expressing a condition on the post-state σ'.

An example of such a lemma is the invariant

$$\begin{array}{l} \vdash \ AuthenticateUserL \Rightarrow uid : \mathsf{dom}(session\_table) \\ \qquad\qquad\qquad\qquad \Rightarrow session\_table'(uid) = session\_table(uid)\,, \end{array}$$

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:

$$\begin{array}{ll} \vdash & (\exists\, DARMA \bullet AuthenticateUserW \land AuthenticateUserL) \lor \\ & (\exists\, DARMA \bullet GenerateSignatureW \land NopOperationL) \lor \\ & (\exists\, DARMA \bullet GenerateSignatureW \land GenerateSignatureL) \lor \\ & (\exists\, DARMA \bullet LogoutW \land LogoutL) \\ \Leftrightarrow & System\,. \end{array}$$

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 assumption over steps in traces by case-splitting: if we have a trace t and a system transition $(s, s') = (t(n), t(n+1))$, a property $P(s, s')$ holds if it holds for the four possible system transitions.

With respect to (3), we proved a large number of global invariants, i.e., formulas of the form $\forall t : traces \bullet INV(t(n), t(n+1))$. 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.

5 Theorem Prove or Model Check?

In previous work [5], we used the SPIN model checker [10] to verify a PROMELA model of the signature architecture. There, we formulated an executable model in terms of synchronously communicating processes, one for each of the different system modules. The requirements were formalized either in linear temporal logic or by augmenting the model (e.g., adding monitor processes) and SPIN was used to verify the result. While there have been other general comparisons of theorem proving versus model checking, e.g., [9], 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, and what is actually measured (see [3] for a discussion of these points). To ensure an accurate comparison, we have kept statistics on both efforts (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 in the formal method.

Figure 5 provides a quantitative comparison of two approaches. We explain the differences below.

Measurement	**PROMELA/SPIN**	**HOL-Z/Isabelle**
Model Variants	4	1
Model Size	647 lines (average)	550 lines
Model Bounds	2 users, 2 sessions	unbounded
Property Size	184 lines	50 lines
Proof Size	none	3662 lines
Property Specification Time	6 days	2 days
System Modeling Time	17 days	12 days
Verification Time	(included above)	19 days
Proof Checking Time	14 hours	12 minutes
Total Time	23 days	33 days
Expert Input Required	10%	60%

Fig. 5. Statistics on the Two Verifications

Size. In PROMELA, we built an initial model of the system, which we adapted afterwards for each of the three properties that we verified. 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.

In contrast, the HOL-Z property specifications are considerably more concise, due to their greater generality. In the PROMELA models, all of the relevant data domains (messages, keys, users, etc.) were bounded to support finite-state model checking. Hence all statements quantifying over these sets must be translated into finite, but large, conjunctions or disjunctions. Moreover, rather than using event predicates as in HOL-Z, we had to formulate state changes in terms of explicit statements about program points as well as manipulated data. This too results in a more voluminous specification. So here we see one of the advantages of working with a general, behavioral model as opposed to a programing language-based (PROMELA) model.

Time. More time was spent in the theorem-proving approach than in the model-checking approach.[2] 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 30%, is not so significant. Moreover, this difference is even less significant when one accounts for the fact that, in the HOL-Z verification, 5 of these days were spent proving stronger formalizations of the properties (see below).

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. This involves tuning constants as well as introducing abstractions and other simplifications. In some cases, the complexity of the model may even increase, due to the addition of auxiliary variables, assertions, and new (monitor) processes. All of these additions were necessary during our verification and hence the need to create three additional model variants, one for each property verified. The time spent with these activities 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

[2] Proof checking times, measuring the times taken by the SPIN and Isabelle systems, are on a 3 gigaherz Pentium IV computer with 1 gigabyte of RAM.

through the Isabelle system, as explained in Section 4.2, 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 simplification, it is certainly 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 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, in other words, there cannot be a logout between these events. This weaker statement (i.e., omitting the last conjunct in the theorem statement in Section 4.1) is what we formalized and verified in SPIN. In HOL-Z, working through the necessary invariants led us to realize that the stronger theorem was actually intended and held.

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. In the SPIN case study, most of the expert help required was in formulating properties (which turned out to be surprisingly tricky) and simplifying the PROMELA models so that SPIN would terminate. 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. While formulating the security properties in Z was possible without expert advice, this was not so with theorem proving, where considerable hands-on work by the expert was necessary. This is reflected by the 60% expert contribution reported in Table 5.

What was modeled and verified. Finally, the numbers given do not reflect that there were substantial differences in what was modeled and verified. A standard benefit of using a rich logic, like HOL-Z, is that one can directly model infinite data domains in their full generality, rather than settling for some finite approximation. This was also the case here, where PROMELA modeling required bounding all of the relevant data domains. Hence, the HOL-Z model is both more general and the theorems proven are significantly stronger.

A more subtle difference stems from the use of a declarative versus an operational approach. In HOL-Z we did not need to commit to either particular data types or concrete procedures for data manipulation. 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 manger 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.

References

1. J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 1996.
2. T. Arai, T. Sekiguchi, M. Satoh, T. Inoue, T. Nakamura, and H. 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.
3. D. Basin and M. Kaufmann. The Boyer-Moore Prover and Nuprl: An experimental comparison. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 90 – 119. Cambridge University Press, 1991.
4. D. Basin, H. Kuruma, K. Takaragi, and B. 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.
5. D. Basin, K. Miyazaki, and K. Takaragi. A formal analysis of a digital signature architecture. In S. Jajodia and L. Strous, editors, *Integrity and Internal Control in Information Systems, IV*, pages 31–48. Kluwer Academic Publishers, 2004.
6. A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, Feb. 2003.
7. A. D. Brucker and B. Wolff. A case study of a formalized security architecture. In *Electronic Notes in Theoretical Computer Science*, volume 80. Elsevier Science Publishers, 2003.
8. C. 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.
9. A. Gupta. Formal hardware verification methods: A survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992.
10. G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
11. International Standard ISO/IEC 13568:2002. Information technology — Z formal specification notation — syntax, type system and semantics.
12. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
13. 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.
14. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International, New Jersey, second edition, 1992.
15. S. Susaki and T. Matsumoto. Alibi establishment for electronic signatures. *Information Processing Society of Japan*, 43(8):2381–2393, 2002. In Japanese.
16. J. Woodcock and J. Davies. *Using Z.* Prentice-Hall International, New Jersey, 1996.