Formal Security Analysis with Interacting State Machines

David von Oheimb and Volkmar Lotz

Siemens AG, Corporate Technology, D-81730 Munich,
{David.von.Oheimb|Volkmar.Lotz}@siemens.com

Abstract. We introduce the ISM approach, a framework for modeling and verifying reactive systems in a formal, even machine-checked, way. The framework has been developed for applications in security analysis. It is based on the notion of Interacting State Machines (ISMs), kind of high-level Input/Output Automata. The ISM framework is used to define system models and present them graphically with the AutoFocus tool, to let them be checked for consistency and translated to a representation within the theorem prover Isabelle/HOL (or alternatively to define them directly as Isabelle theory sections), and finally to employ the theorem prover for performing any kind of syntactic and semantic checks, in particular semi-automatic verification. We demonstrate that the framework can be fruitfully applied for formal system analysis by two classical application examples: the LKW model of the Infineon SLE 66 smart card chip and Lowe's fix of the Needham-Schroeder Public-Key Protocol.

Keywords: security, formal analysis, Interacting State Machines, Isabelle/HOL, AutoFocus, smart cards, protocols

1 Introduction

1.1 Motivation

In industrial environments, there is an increased demand for rigorous analysis of security properties of systems. Due to restrictions imposed by the application domain, the system environment, and business needs, new security mechanisms and architectures have to be invented frequently, with time-to-market pressure and intellectual property considerations obstructing the chance to gain confidence by exposing a proposed solution to the security community (which has been shown to be appropriate for cryptographic algorithm assessment). Formal analysis of suitable abstractions of systems has instead turned out to be extremely helpful in reasoning about a system's security, since the mathematical precision of the arguments allows for maximal confidence in the results obtained and, thus, in the security of the system being modeled.

The importance of formal analysis – on top of open review – in security assessment is, for instance, reflected by the requirements stated for high assurance levels of criteria like ITSEC [ITS91] and CC [CC99], which include formal security modeling and formal system development steps, and the achievements of the security protocol verification community, which discovered flaws in protocols that failed to be detected by informal approaches.

However, even in a formal setting it is easy to make – minor and sometimes even major – mistakes: undefined expressions, type mismatches, inconsistent specifications, missing evidence in proofs, false conclusions etc. Therefore, pure pen-and-paper formalizations cannot be considered fully reliable. Machine-checking of formal objects and structures has to be employed in order to significantly reduce the occurrence of such mistakes. Machine support additionally gives the opportunity to represent and deal with formal objects – both specifications and proofs – in an easy-to-comprehend way, which is a prerequisite for introducing formal approaches in an industrial environment characterized by time and cost restrictions.

1.2 Goals

A framework for machine-assisted formal security analysis that is particularly suited for industrial use should enjoy a number of properties:

Expressiveness. It should be possible to describe any typical security sensitive computation, storage, and communication system in an abstract way. This requires in particular the notions of state transformation, concurrency, and message passing.

Flexibility. Since IT systems and their security threats evolve quickly, the models produced within the framework should be easily adaptable and extendable as necessary to reflect the changes.

Simplicity. Modeling a system, stating its properties and proving them should require as little expertise and time as possible while maintaining the rigor of a fully formal approach.

Graphical capabilities. System models should be representable as diagrams that provide a good overview of the system structure and advance a quick intuition about its behavior.

Maturity of the semantics. The specification formalism should build upon a well-understood logic and have a well-defined semantics that supports reasoning about, e.g., invariants and refinement.

Availability of tools. The framework should be built from existing widely available (open-source) software like editors and proof tools and require at most minor modifications or extensions to them.

Since we did not find an existing framework that fulfills all these requirements to a satisfactory extent, we decided to build our own.

1.3 Related Work

The *IOA Language and Toolset* [GL98,Kay01] is a framework for analyzing computational processes with aims very similar to ours. It consists of a specification language and tool support for simulation, theorem proving, model checking, and code generation, where by now the simulation aspect is developed most and theorem proving support is limited to PVS. Its semantic foundation is the notion of *I/O Automata (IOAs)* [LT89] modeling asynchronous distributed computation with synchronous communication. Since the notion is based on transition systems augmented by communication primitives (rather than e.g. a process algebra augmented by local computation primitives), it is fairly easy to understand. It is equipped with a well-developed meta theory supporting refinement and compositional reasoning. System properties, both safety and liveness ones, may be described using temporal logics and proved by model checking and interactive theorem proving.

The only — but severe — drawback of IOAs from our perspective, in particular when modeling system security in an abstract way, is that their interaction scheme is rather low-level: buffered communication has to be modeled explicitly, and transitions involving several related input, internal processing, and output activities cannot be expressed atomically. Instead, each high-level transition has to be split into multiple low-level transitions, and between these, any number of further input events may take place due to the input-enabledness of IOAs. The solution to this problem is to add input buffers that accumulate messages asynchronously. An automaton may retrieve messages from multiple buffers, process them and send output to multiple buffers, and all this can be done simultaneously within a single atomic[1] transition. Our notion of ISMs, first described in [Ohe02], provides for that.

A further related framework that provided inspiration for ours is AutoFocus [HSSS96] – see §2.2 for more details. Even though developed primarily for modeling and verifying functional properties of embedded systems, it is used also for the security analysis of general distributed systems [WW01,JW01].

Other related approaches combine state-oriented and message-oriented description methods, for example translating CSP to B [But99] or Z to CSP [Fis00]. The drawback of such hybrids is that the user has to deal with two different nontrivial formalisms. Moreover, theorem proving support respecting the structure of the mixed-style specifications seems not to be available.

2 Preliminaries

In this section, we briefly introduce the two software tools we rely on and comment on their suitability for the ISM approach.

[1] Even though these high-level transitions are atomic, the corresponding I/O events are independent of each other because of the buffered asynchronous output semantics; thus there is no need for action refinement.

2.1 Isabelle/HOL

Isabelle [NPW02] is a generic theorem prover that has been instantiated to many logics, in particular the very practical *Higher-Order Logic (HOL)*. Isabelle/HOL [PNW+] is a predicate logic based on the simply-typed λ-calculus and thus in a sense combines logical and functional programming. Being quite expressive and supporting automatic type inference, it is the most important and best supported logic of Isabelle. The lack of dependent types introduces a minor nuisance for applications like ours: for systems consisting of more than one ISM, there has to be a single type of message contents into which all message data is injected, and analogously for the local states of the automata composed in parallel.

Proofs are conducted primarily in an interactive fashion where automatic and semi-automatic methods are available to tackle the routine parts. The Isabelle system is well-documented and well-supported, is freely available (including sources) and comes with the excellent user interface Proof General [AGKS99]. We consider it the most flexible and mature verification environment available. Using Isabelle/HOL, security properties can be expressed easily and adequately and verified with powerful proof methods.

2.2 AutoFocus

AutoFocus [HSSS96] is a freely available specification and simulation tool for distributed systems. Components and their behavior are specified by a combination of *System Structure Diagrams (SSDs)*, *State Transition Diagrams (STDs)* and auxiliary *Data Type Definitions (DTDs)*. Their execution can be visualized using *Extended Event Traces (EETs)*. Various back-ends including code generators and interfaces to model checkers may be acquired by purchase from Validas [S+].

We employ AutoFocus for its strengths concerning graphical design and presentation, which is important when setting up models in collaboration with clients (where strong familiarity with formal notations cannot be assumed), when documenting our work, and publishing its results. For abstract security modeling, there are currently two problems with AutoFocus. First, expressiveness is limited concerning the type system and the handling of underspecification. Second, due to the original emphasis of AutoFocus on embedded systems, the underlying semantics is still clock-synchronous. In contrast, for the most of our applications, an asynchronous (buffered) semantics is more adequate, which is under consideration also for future versions of AutoFocus. Using an alternative semantics implies that we cannot make use of the simulation, code generation and model checking capabilites of current AutoFocus and its back-ends. Yet this is not a real obstacle for us since we are interested mainly in its graphic capabilities and the offered specification syntax is general enough to cover our deviating semantics as well.

3 The ISM approach

ISMs are the core of our modeling and verification framework. In this section we explain the ISM concepts and semantics both in an intuitive way and as rigorous mathematical definitions. Moreover, we comment briefly on the ISM representation in AutoFocus and define the syntax of ISM sections in Isabelle/HOL theories. In the subsequent sections we present two classical case studies.

We use ISMs as building blocks for defining system models of a wide range of IT systems and expressing and verifying their security properties. At the time of writing, we have applied the ISM formalism in three major projects. They include the analysis of a complex database access control system for Siemens Medical Solutions and of the Infineon SLE88 smart card processor memory management [OLW04]. More information on the current status of the ISM framework, including the sources, a manual, and all publications, may be found at the project home page, `http://ddvo.net/ISM/`.

The ISM formalism has been extended to include global state [OL03]. This can be used, for instance, to provide for dynamic activation state and communication topology [OL03] or ambient-like administrative domains [KO03] or even their combination [KO03].

3.1 Concept of Interacting State Machines

An *Interacting State Machine (ISM)* is an automaton whose state transitions may involve multiple input and output simultaneously on any number of ports. As the name suggests, the key concepts of ISMs are states (and in particular the transitions between them) and interaction. By *interaction* we mean explicit buffered communication via named ports (which are also called connections), where on each *port*, (typically) one receiver listens to possibly many senders. Figure 1 gives the basic ISM structure.

Any number of ISMs may be composed in parallel by interleaving their transitions and forming I/O connections among peer ISMs. The local state of the resulting ISM is essentially the Cartesian product of the local states of its components. The top-level composition is called an ISM *system*. In [OL03] we extend the ISM concept by the notion of global state, which is not directly visible to ISMs but can control the whole system structure. The global state is affected by commands contained in transitions of elementary ISMs.

A *configuration* of an ISM consists of its input buffer state and local state. The *local state* may have arbitrary structure but typically is the Cartesian product of a *control state* which is of finite type and a *data state* which is a record of named fields representing local variables. Each ISM has a single[2] local *initial state*.

[2] If a non-singleton set of initial states is required, this may be simulated by nondeterministic spontaneous transitions from a single dummy initial state.

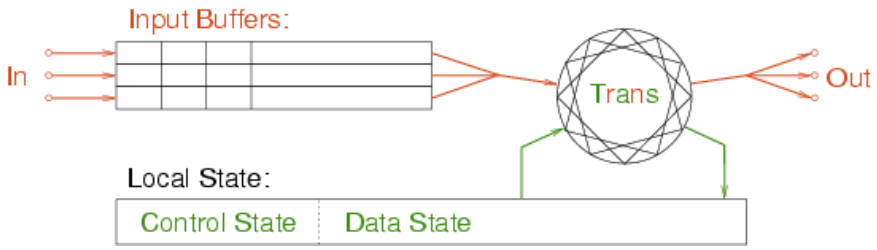

Fig. 1. ISM structure

The input buffers of an ISM are a family of (unbounded) message FIFOs, indexed by port names. The buffers are not part of elementary ISMs but are introduced by the parallel composition. Input ports can – but in most applications should not – be shared among ISMs, which leads to nondeterministic competition on each input item, without fairness guarantees.

Message exchange is triggered by an output operation of any ISM within the system. Input from the environment may be modeled with suitable ISMs. Inputs cannot be blocked, i.e. they may occur at any time, appending the received value to the corresponding FIFO. Values stored in the input buffers related to an ISM are received and processed by the ISM when it is ready to do so.

The actions of ISMs are given as user-defined *transitions*, which may be nondeterministic and can be specified in any relational style. Thus for each transition the user has the choice to define it in an operational (i.e., executable) or axiomatic (i.e., property-oriented) fashion or a mixture of the two. Transition rules specify that – potentially under some precondition that typically includes matching of messages in the input buffers – the ISM consumes some input, makes a local state transition, and produces some output. The output is appended to the respective input buffers specified by port names. Direct or indirect feedback is possible. Multicast is not directly supported but may be explicitly modeled easily.

An ISM system *run* is any prefix of the sequence of configurations reachable from the initial configuration. The length of a run is not bounded but finite. Finiteness allows for a simple trace semantics, but on the other hand implies that we cannot handle liveness properties. Yet we do not feel this as a real restriction because most relevant properties are essentially safety properties: practical guarantees about the existence of future events typically involve timeouts.

Transitions of different ISMs that are composed in parallel cannot directly interfere with each other but are related only by the causality wrt. the messages interchanged. Execution gets stuck (i.e., deadlocks) when there is no component that can perform any step. As is typical for reactive systems, there is no built-in notion of final or accepting states.

3.2 ISM Semantics

This subsection gives the logical meaning of ISMs, which is both an extension and a slight simplification of the definitions given in [Ohe02]. As the modifications pervade all parts of the ISM definitions, and for self-containedness, it appears mandatory to rephrase all of them.

First some general remarks on the presentation: all definitions and proofs have been developed as a hierarchy of Isabelle/HOL theories and machine-checked using this tool. One important effect of this approach is that many kinds of mistakes like type mismatches can be ruled out. Using the LaTeX documentation feature of Isabelle would even preclude typographic slips in the presentation but on the other hand would introduce some technicalities many readers would not be familiar with. Therefore, we give the semantics in traditional "mathematical" style in order to enhance readability. We sometimes make use of λ-abstraction borrowed from the λ-calculus, but write (multi-argument) function application in the conventional form, e.g. $f(a, b, c)$. Occasionally we make use of partial application (aka. *currying*), such that, in the example just given, $f(a, b)$ is an intermediate function that requires a third parameter before yielding the actual function result.

Message Families Let $\mathcal{M}$ be the type of all messages potentially exchanged by ISMs and $\mathcal{P}$ the type of port names. Then the *message families*, which are used to denote both input[3] buffers and input/output patterns, have type $MSGs = \mathcal{P} \to \mathcal{M}^*$ where $\mathcal{M}^*$ is any finite sequence of elements of $\mathcal{M}$. We will make use of the following operations on message families:

- the term $\not\square$ denotes the empty message family $\lambda p.\ \langle\rangle$ where $\langle\rangle$ denotes the empty sequence
- the term $mdom(m)$ abbreviates $\{p.\ m(p) \neq \langle\rangle\}$, i.e. the domain of m
- the infix operation .@. concatenates two message families m and n pointwise: $(m\ .@.\ n)(p) = m(p)\ @\ n(p)$

States and Transitions A set of ISM transitions has type $TRANS(\Sigma) = \wp((MSGs \times \Sigma) \times (MSGs \times \Sigma))$ where the parameter Σ stands for the type of the local state and the two occurrences of $MSGs$ stand for input and output patterns, respectively. Each element has the form $((i, \sigma), (o, \sigma'))$ and means that the ISM can (possibly nondeterministically) perform a step from local state σ to σ', consuming input i and producing output o. Simultaneous input and/or output on multiple channels can be specified because both i and o each denote whole message families. In contrast to the original definition of ISMs [Ohe02], within a transition, input is described by patterns of messages consumed in the given step — not by a transition between the state of the input buffer before and after the transition. This simplifies the definition of single ISMs and shifts the concept of input buffering to the places where it is indispensable: at the definitions of parallel composition and automata runs.

[3] Recall that output buffers are not required.

Elementary ISMs An ISM is given as a quadruple[4] $a = (In(a), Out(a), \sigma_0(a), Trans(a))$ of type $ISM(\Sigma) = \wp(\mathcal{P}) \times \wp(\mathcal{P}) \times \Sigma \times TRANS(\Sigma)$ where

- $In(a)$ is the set of input port names
- $Out(a)$ is the set of output port names
- $\sigma_0(a)$ is the initial local state
- $Trans(a)$ is the transition relation

Such an ISM is *well-formed* iff all the port names actually used in the transitions for input or output respect the I/O interface of the ISM, i.e. $ipns(a) \subseteq In(a)$ and $opns(a) \subseteq Out(a)$ where

- $ipns(a) = \bigcup_{t \in Trans(a)} mdom((\lambda((i,\sigma),(o,\sigma')).\ i)(t))$
- $opns(a) = \bigcup_{t \in Trans(a)} mdom((\lambda((i,\sigma),(o,\sigma')).\ o)(t))$

Note that $In(a)$ and $Out(a)$ may overlap, which allows for direct feedback within parallel composition.

Runs Below we will define composite ISM runs, i.e. the parallel composition and execution of a family of ISMs, directly in one step. Nevertheless, we first define the two notions of ISM runs and parallel composition independently. Defining parallel composition in isolation not only makes it easier to understand but also enables hierarchical analysis and design.

The *open runs* of an ISM a, denoted by $Runs(a) \in \wp(\Sigma^*)$, are finite sequences of states that are inductively defined as

$$\frac{}{\langle \sigma_0(a) \rangle \in Runs(a)}$$

$$\frac{ss^\frown\sigma \in Runs(a) \quad ((i,\sigma),(o,\sigma')) \in Trans(a)}{ss^\frown\sigma^\frown\sigma' \in Runs(a)}$$

The operator $^\frown$ appends elements to a sequence.

This form of runs is called *open* because in each step the environment provides arbitrary input to the ISM, and any output of the ISM is discarded. If feedback from output to input is desired, one can achieve this by applying the parallel composition operator to the singleton family of ISMs consisting just of a, described next.

Parallel Composition Any number of ISMs can be combined in parallel to form a single composite ISM, which may be further combined with others, etc. By identifying input and output buffers of ISMs to be combined, internal communication including feedback loops can be introduced as shown in Figure 2.

[4] The definition pattern $x = (sel_1(x), sel_2(x), \ldots)$ should not be understood as a recursive definition of x but as a shorthand introducing a tuple with typical name x and with selectors (i.e., projection functions) sel_1, sel_2, ...

The *parallel composition* $\|_{i\in I} A_i$ of a family of ISMs $A = (A_i)_{i\in I}$ is an ISM of type $ISM(CONF(\Pi_{i\in I}\Sigma_i))$ where I is any index set I and for any X, the type of an ISM *configuration* $CONF(X)$ is defined as $MSGs \times X$. Here $MSGs$ stands for the type of internal buffers. The composite ISM is defined as the quadruple $(AllIn(A)\backslash AllOut(A),\ AllOut(A)\backslash AllIn(A),\ (¤, S_0(A)),\ PTrans(A))$ where

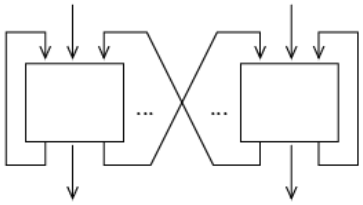

Fig. 2. General communication pattern within parallel composition

- $AllIn(A) = \bigcup_{i\in I} In(A_i)$
- $AllOut(A) = \bigcup_{i\in I} Out(A_i)$
- $¤$ gives the initial value of the internal buffers, which are used to handle I/O among peers as well as direct feedback
- $S_0(A) = \Pi_{i\in I}(\sigma_0(A_i))$ is the Cartesian product of all initial local states
- $PTrans(A)$ of type $TRANS(CONF(\Pi_{i\in I}\Sigma_i))$ is the parallel composition of their transition relations.

The pre- and post-states in the composed transition relation refer not only to the Cartesian product of all local states but also to a message family b. As already mentioned above for the initial state, the role of b is to buffer internal I/O. Apart from this, the composed transition relation is defined simply as the interleaving of the transitions of the component ISMs:

$$\frac{\begin{array}{c} j \in I \\ ((i,\sigma),(o,\sigma')) \in Trans(A_j) \end{array}}{\begin{array}{l} ((i_{|\overline{AllOut(A)}}, (i_{|AllOut(A)}\ .@.\ b, S[j{:=}\sigma\,])), \\ \ (o_{|\overline{AllIn(A)}},\ (b\ .@.\ o_{|AllIn(A)},\ \ S[j{:=}\sigma'])))\ \in PTrans(A) \end{array}}$$

where

- $S[j{:=}\sigma]$ denotes the replacement of the j-th component of the tuple S by σ
- $m_{|P}$ denotes the restriction $\lambda p.$ if $p \in P$ then $m(p)$ else $\langle\rangle$ of the message family m to the set of ports P
- $i_{|\overline{AllOut(A)}}$ denotes those parts of the input i provided not by the output of peer ISMS but by outer ISMs
- $i_{|AllOut(A)}$ denotes the internal input from peer ISMs or direct feedback, which is taken from the current buffer contents b
- $o_{|\overline{AllIn(A)}}$ denotes those parts of the output o provided to outer ISMs
- $o_{|AllIn(A)}$ denotes the internal output to peer ISMs or direct feedback, which is added to the current buffer contents b.

A parallel composition is *well-formed* iff the inputs of the individual components do not overlap: $\forall i\ j.\ i \neq j \longrightarrow In(A_i) \cap In(A_j) = \emptyset$. On the other hand, outputs may overlap, which allows the outputs of different ISMs to interleave nondeterministically.

A family A of ISMs is called *closed* iff $AllIn(A) = AllOut(A)$, i.e. there is no interaction with any outside ISMs. If a system is modeled with a closed ISM family and input from the environment is important, this may be modeled with an ISM that belongs to the family and does nothing but generating all possible input patterns.

When composing ISMs, it is occasionally necessary to prevent name clashes or to hide connections, which can be achieved by suitable renaming of ports.

Composite Runs We define ISM runs not only for single (possibly composite) ISMs but also directly for closed families of ISMs intended to run in parallel. The above definition of parallel composition may be used in combination with composite runs to describe inner (possibly nested) levels of parallel composition.

The set of all possible *composite runs* is denoted by $CRuns(A)$ and has type $\wp((CONF(\Pi_{i\in I}\Sigma_i))^*)$ corresponding to the ISM type $ISM(\Pi_{i\in I}\Sigma_i)$. Its elements are finite sequences of configurations, inductively defined as

$$\frac{}{\langle(\not\square, S_0(A))\rangle \in CRuns(A)}$$

$$\frac{\begin{array}{c} j \in I \\ cs^\frown(i\ .@.\ b, S[j:=\sigma]) \in CRuns(A) \\ ((i,\sigma),(o,\sigma')) \in Trans(A_j) \end{array}}{cs \frown (i\ .@.\ b, S[j:=\sigma]) \frown (b\ .@.\ o, (S[j:=\sigma'])) \in CRuns(A)}$$

Traces of composite runs have the form $\langle(\not\square,S_0(A)),\ (b_1,S_1),\ (b_2,S_2),\ \ldots\rangle$ where each element of the sequence is a pair of the current internal buffer contents and the Cartesian product of all the currently relevant local states.

One can show that composite runs of any closed family A of well-formed ISMs are equivalent to the runs of the parallel composition of the same family: $wf\_isms(A)\ \wedge\ closed(A)\ \longrightarrow\ Runs(\|_{i\in I}A_i) = CRuns(A)$.

3.3 AutoFocus representation

By design, ISMs have almost the same structure as the automata definable with AutoFocus [HSSS96], and thus we can use AutoFocus as a graphical front-end to our Isabelle implementation. We will employ AutoFocus diagrams when introducing the application examples below.

In a typical application of our framework, ISMs are first specified[5] as standard non-hierarchical AutoFocus automata, saved in the so-called *Quest* file format, and then translated into suitable Isabelle theory files by a tool program [Nan02,ON02].

[5] see the online tutorial `http://autofocus.in.tum.de/nelli/englisch/html/`

3.4 Isabelle representation

An ISM section is introduced by the keyword **ism** and has the following general structure[6]:

ism *name* *((*(*param_name* **::** *param_type*)*)** =
  ports *pn_type*
    inputs *I_pns*
    outputs *O_pns*
  messages *msg_type*
  states *[state_type]*
  *[***control** *cs_type [***init** *cs_expr0]]*
  *[***data** *ds_type [***init** *ds_expr0] [***name** *ds_name]]*
 *[***transitions**
  *(tr_name [attrs]***:** *[cs_expr* `->` *cs_expr']*
  *[***pre** *(bool_expr)*$^+$*]*
  *[***in** *([***multi***]** *I_pn I_msgs)*$^+$*]*
  *[***out** *([***multi***]** *O_pn O_msgs)*$^+$*]*
  *[***post** *((lvar_name* **:=** *expr)*$^+$ *| ds_expr')]*
)$^+$*]*

The meaning of the individual parts is as follows.

- The ISM definition will be referred to by *name*. It may have any number of parameters, each declared by *param_name* and a corresponding *param_type*. The parameters may be used throughout the definition body.
- The type expression *pn_type* gives the Isabelle/HOL type of the port names, while *I_pns* and *O_pns* denote the set of input and output port names, respectively.
- The type expression *msg_type* gives the type of the messages, which is typically an algebraic datatype with a constructor for each kind of message.
- The optional *state_type* should be given if the current ISM forms part of a parallel composition and the state types of the ISMs involved differ. In this case, *state_type* should be a free algebraic datatype with a constructor for each state type of the ISMs involved.
  The type expressions *cs_type* and *ds_type* give the types of the control and data state, respectively, while the optional terms *cs_expr0* and *ds_expr0* specify their initial values — if not given, they default to some arbitrary value. Either (i.e., not both) the control state or the data state may be absent.
  The optional logical variable name *ds_name*, which defaults to `s`, may be used to refer to the whole data state within transition rules.

Transitions are given via named rules where *attrs* is an optional list of attributes, e.g. **[intro]**. The control states (if any) before and after the transition are specified by the expressions[7] *cs_expr* and *cs_expr'*.

[6] *[...]* marks optional parts, *(...)*$^+$ means one or more comma-delimited occurrences

[7] These need not be constant but may contain also variables, which is useful for modeling generic transitions. In this case, one such transition has to be represented by a set of transitions within AutoFocus.

Expressions within a rule may refer to the logical data state variable mentioned above. In particular, assuming that `s` is the name of the data state variable, then the value of any local variable `lvar` of the ISM may be referred to by `lvar s`. The scope of free variables appearing in a rule is the whole rule, i.e. free variables are implicitly universally quantified (immediately) outside each rule. All the following parts of a transition rule are optional:

- The **pre** part contains guard expressions *bool_expr*, i.e. preconditions constraining the enabledness of a transition.
- The **in** part gives input port names (or sets of them if preceded by **multi**) *I_pn*, each in conjunction with a list *I_msgs* of message patterns expected to be present in the corresponding input buffer(s). When an ISM executes a transition, any free variables in message patterns are bound to the actual values that have been input. Each port names should appear at most once within a **in** part. Any input port not explicitly mentioned is left untouched.
- The **out** part gives output port names *O_pn*, each in conjunction with an expression *O_msgs* denoting a list of values designated for output to the corresponding port. The variant using **multi** is used to specify multicasts. Each port name should be used at most once within each **out** part. Any output port not mentioned does not obtain new output.
- The **post** part describes assignments of values *expr* to the local variables *lvar_name* of the data state. Variables not mentioned remain invariant. Alternatively, an expression *ds_expr'* may be given that represents the entire new data state after the transition. Assignments to the local variables suit an operational style, whereas an axiomatic style can be achieved using *ds_expr'* (in conjunction with suitable constraints in the preconditions).

An **ism** theory section is translated to Isabelle/HOL concepts in a straightforward way using an extension to Isabelle, as described in [Nan02]. In particular, each ISM section is translated to a record definition with the appropriate fields, the most complex one being the transition relation, which is defined via an inductive (but not actually recursive) definition.

The meta theory of ISMs that we have defined in Isabelle/HOL includes all concepts mentioned in §3.2, in particular well-formedness, renaming, parallel composition, runs, and composite runs. Further auxiliary concepts are introduced as well, in particular reachability and induction schemes related to ISM runs. The characteristic properties of these concepts, as required for system verification, are derived within Isabelle/HOL. All details of the meta theory may be found in [ON02].

Example **ism** sections will be given in §4.4 and §5.2.

4 LKW Model for the Infineon SLE 66

We give a slightly extended and improved version of the LKW formal security model for the Infineon SLE 66 smart card processor.

4.1 The SLE 66 family

SLE 66 is the short name of a family of smart card chips by Infineon. Each chip consists of a CPU including an encryption unit, RAM, ROM, and EEPROM, which stores e.g. firmware and personalization data.

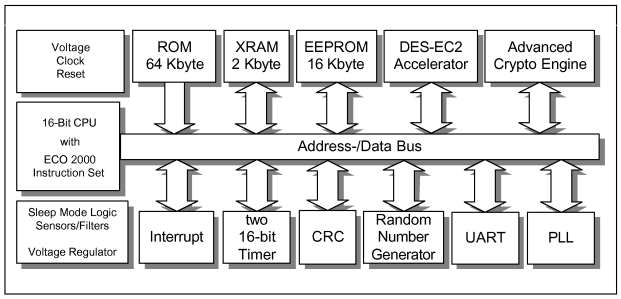

Fig. 3. SLE 66 Block Diagram

The chip has been designed as a general-purpose microprocessor with special hardware supporting security-sensitive applications like electronic passport or payment systems. In contrast to the successor family, SLE 88, these processors do not provide separation of memory via a MMU [OLW04] or any operation system functionality but provide a secure platform for a customized BIOS and essentially a single application. Therefore, security has to be dealt with at a very elementary level where nothing can be assumed about higher-level functionality.

The most important security objective is to preserve the security of information stored in the memory components. In more detail:

- The data items stored in any of the memory components shall be protected against unauthorized disclosure or modification.
- The security relevant functions stored in ROM or EEPROM shall be protected against unauthorized disclosure or modification.
- Hardware test routines shall be protected against unauthorized execution.

The objectives are achieved by implementing a set of security enforcing functions which mainly perform the following two tasks:

- The system passes several phases during its lifetime. Entry to the phases is controlled by *test functions*, which check different flags and give a specified level of authorization.
- Additionally, all data stored in the memory components is encrypted by hardware means, utilizing several keys and key sources with a chip specific random number among them.

4.2 LKW formal security model

The *LKW model* [LKW00] has been one of the first formal models for security properties of hardware chips. It has been used very successfully within the security evaluation process for the whole SLE 66 family on ITSEC level E4 high and the corresponding Evaluation Assurance Level 5 (*semi-formally designed and tested*, which includes a formal security model) [CC99]. A slight extension has been introduced [OLW02] in order to reflect additional application-oriented security objectives defined in the Smart Card IC Platform Protection Profile [AHIP01]. More recently, we have added an analysis of nonleakage [Ohe04].

Developing the original LKW model took about two months of work, including understanding and discussing the system design and security target, investigating modeling alternatives, discussing the model with the chip developers, and supporting the evaluation process. The formal parts made up about ten percent of the whole evaluation and certification effort which was even based on existing development documents. Re-stating the model with the ISM approach took about two weeks. Incorporating the extension mentioned above took just a few days including discussions etc. These numbers may serve as an indicator for estimating formal modeling efforts in future evaluation processes.

Meanwhile we have developed also a security model of the SLE 88 memory management unit [OWL03] following the ISM approach as well.

The formal security policy model of the SLE 66 consists of two parts: a system model describing the processor's behavior on an abstract level by means of a state transition automaton with input and output, and a set of security objective specifications given as properties of automata runs. Thus one can prove that the security objectives are met by the system model. Interpreting the system model in terms of the real processor then allows one to conclude with some evidence that the processor indeed meets its security objectives as required by ITSEC E4 assessment criteria.

The style of the LKW security model is ad-hoc, but using classical formal access control models instead would not be appropriate because they introduce notational overhead that would not be justified in the context of the SLE 66 evaluation and because they are not flexible enough to handle phase transitions and the like adequately.

The LKW model has been done originally as a pen-and-paper work, i.e. without tool assistance. Inevitably, even fully reviewed descriptions of the model contained many (mostly minor) syntactical, typographical and semantical slips as well as type errors, but also omissions like missing assumptions and incomplete proofs. Therefore it was desirable to formalize the model in a machine-checked way, applying a well-developed meta theory. At first, using the Isabelle implementation of IOAs [Mül98] for this purpose seemed promising, yet the weak structure of IOA transitions appeared inappropriate, which became one of our motivations to invent ISMs. Using the ISM approach, the LKW model can be represented adequately and with maximal quality, as demonstrated on the following pages.

4.3 AutoFocus Diagrams

On the abstract level of the LKW model, the system architecture of the SLE 66 is rather trivial: there is one component with one input port named `In` and one output port named `Out`, as depicted by Figure 4. The data state of the component consists of two stores mapping names of functions to the corresponding function code and data objects to corresponding data values.

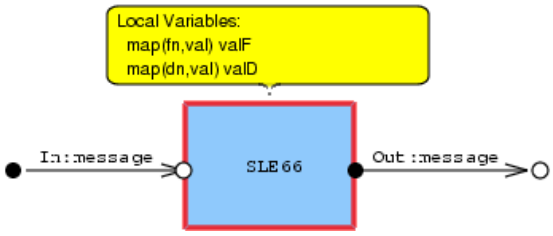

Fig. 4. SLE66 System Structure Diagram

Much more involved is the structure of the state transitions. There are four control states corresponding to the *phases* of the SLE 66 life cycle:

Phase *0*: construction of the chip
Phase *1*: upload of Smartcard Embedded Software and personalization
Phase *2*: normal usage
Phase *Error*: locked mode from which there is no escape

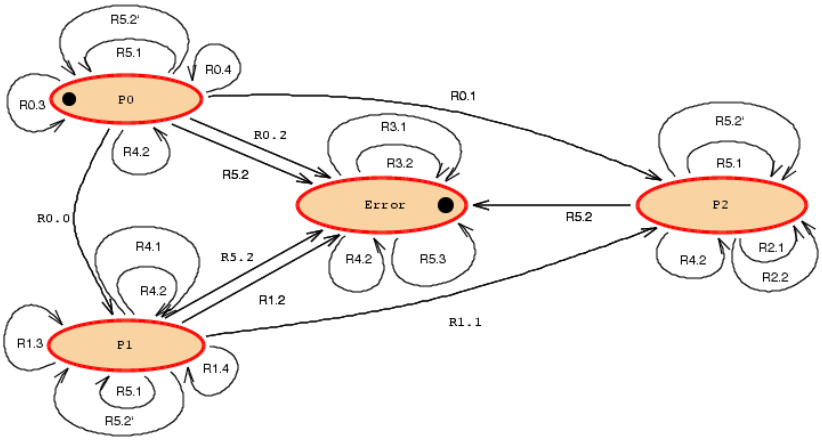

Fig. 5. SLE 66 State Transition Diagram

In order to keep the state transition diagram clear, Figure 5 contains all control states and transitions, but instead of showing the preconditions, inputs, outputs, and changes to the data state, we just label the transitions with the names of the corresponding transition rules. These are described in detail in §4.4, while here we give an informal general description:

R0.0 thru R0.4 describe the execution of functions in the initial phase 0. Only the processor manufacturer is allowed to invoke functions in this phase and the requested function must be present.

R0.0 states that if the function belongs to class `FTest0` and the corresponding test succeeds, phase 1 will be entered, and the test functions of that class are disabled.

R0.1 describes a shortcut leaving out phase 1: if the function belongs to class `FTest1` and the test succeeds, phase 2 will be entered, and all test functions are disabled.

R0.2 states that if a test fails, the system will enter the error state.

R0.3 models the successful execution of any other function, in which case the function may change the chip state and yield a value.

R0.4 states that in all remaining cases of function execution, the chip responds with `No` and its state remains unchanged.

R1.1 thru R1.4 describe the execution of functions in the upload phase 1 analogously to R0.1 thru R0.4.

R2.1 and R2.2 describe the execution of functions in the usage phase 2 analogously to R0.3 and R0.4.

R3.1 and R3.2 describe the execution of functions in the error phase analogously to R0.3 and R0.4, except that the only function allowed to be executed in this phase is chip identification.

R4.1 and R4.2 describe the effects of a specific operation used for uploading new (operating system and application) functionality on the chip. This must be done by subjects trusted by the processor manufacturer and is allowed only in phase 1.

R4.1 describes the admissible situations, and

R4.2 describes all other cases.

R5.1 thru R5.3 describe the effects of attacks. Any attempts to tamper with the chip and to read security-relevant objects via physical probing on side channels (by mechanical, electrical, optical, and/or chemical means), for example differential power analysis or inspecting the silicon with a microscope, are modeled as a special "spy" input. Note that modeling physical attacks in more detail is not feasible because this would require a model of physical hardware. In particular, the conditions (and related mechanisms) under which the processor detects a physical attack are beyond the scope of the model.

R5.1 describes the innocent case of reading non-security-relevant objects in any regular phase, which actually reveals the requested information.

R5.2 describes the attempt to reading security-relevant objects in any regular phase. The chip has to detect this and enters the error phase, while the requested object may be revealed or not. This concept is called "destructive reading": one cannot rule out that attacks may reveal information even about security-relevant objects, but after the first of any such attacks, the processor hardware will be "destroyed", i.e. cannot be used regularly.

R5.3 states that in the error phase no (further) information is revealed.

4.4 Isabelle Definition

We describe in detail our ISM model of the SLE 66, which is based on the original LKW model plus the slight extension introduced in [OLW02]. We do this employing the automatic LaTeX documentation facility of Isabelle that can be used like a "literal programming" environment: the user augments an Isabelle theory (in this case representing our SLE 66 model) with comments and other text sections in LaTeX format that may refer (via a special quotation mechanism) to the type declarations, constant definitions, theorems, etc. When Isabelle processes the theory, it generates LaTeX output for all parts of the theory that are marked as relevant for documentation and merges them with the chunks of text supplied by the user. The great advantage of this approach is that the theory (and proof) development and its documentation are always with each other and mistakes typically resulting from typesetting formulas with LaTeX manually are avoided.

The Isabelle theory sources, including the documenting text, may be obtained from [ON02]. For the original description of the LKW model containing, among others, a more general discussion on the benefits of formal modeling, refer to [LKW00].

theory *SLE66 = ISM_package:* — we build on the general ISM definitions

First we have to define a bunch of entities (types, logical constants, etc.) acting as building blocks for the actual ISM theory section. In order to keep the model as abstract as possible, which makes it less bulky to read and simplifies the proofs, we often use underspecification. This important modeling technique means that for part of the types and constants we do not give full definitions but only declarations of their names. We even do not make the encryption of data in the memory components explicit.

Names Objects stored on the chip may be either functions or data and are referred to by object names. The type of these names, *on*, is the disjoint sum of function names *fn* and data object names *dn*, which are not further specified:

typedecl *fn* — function name
typedecl *dn* — data object name
datatype *on = F fn | D dn* — object name

Objects are classified as security-relevant (demanding secrecy and integrity) by including their names in the sets *F_Sec* or *D_Sec*, whose disjoint union is called *Sec*. In order to meet the additional requirements of [AHIP01], the domain of security relevant functions *F_Sec* of the original LKW model has been refined to the disjoint union of *F_PSec* and *F_ASec*, which control the protection of the processor and application functionality, respectively.

In the following theory sections, we declare a list of constants together with their types. We define only part of them, and for part of the remaining ones we give the essential properties in the form of axioms:

consts
```
  f_SN   :: "fn"      — the name of the function giving the serial number
  FTest0 :: "fn set"  — the names of test functions of phase 0
  FTest1 :: "fn set"  — the names of test functions of phase 1
  FTest  :: "fn set"  — the names of all test functions
  F_Sec  :: "fn set"  — the names of all security-relevant functions
  F_PSec :: "fn set"  — the subset of F_Sec relevant for the processor
  F_ASec :: "fn set"  — the names of F_Sec relevant for applications
  F_NSec :: "fn set"  — the names of all non-security-relevant functions
  D_Sec  :: "dn set"  — the names of all security-relevant data objects
  D_PSec :: "dn set"  — the subset of D_Sec relevant for the processor
  D_ASec :: "dn set"  — the names of D_Sec relevant for applications
  D_NSec :: "dn set"  — the names of all non-security-relevant data objects
    Sec  :: "on set"  — the names of all security-relevant objects
```
defs
```
  FTest_def:  "FTest  ≡ FTest0 ∪ FTest1"
  F_ASec_def: "F_ASec ≡ F_Sec - F_PSec"
  D_ASec_def: "D_ASec ≡ D_Sec - D_PSec"
  F_NSec_def: "F_NSec ≡ -F_Sec"
  D_NSec_def: "D_NSec ≡ -D_Sec"
  Sec_def: "Sec ≡ {F fn |fn. fn ∈ F_Sec} ∪ {D dn |dn. dn ∈ D_Sec}"
```
axioms
```
  FTest01_disjunct: "FTest0 ∩ FTest1 = {}"
  f_SN_not_FTest:   "f_SN ∉ FTest"
  F_PSec_is_Sec:    "F_PSec ⊆ F_Sec"
  FTest_is_PSec:    "FTest ⊆ F_PSec"
```

State The abstract state of an SLE 66 chip is a pair, where the first component is the phase in the processor life cycle:

datatype `ph = P0 | P1 | P2 | Error`

We introduce the type `val` for any values, i.e. function code or data stored or processed by the chip. The only thing we need to know about the type `val` is that the serial number of the chip belongs to it.

typedecl `val` — data and function values
consts `SN :: val` — serial number

The second state component is a record of two partial functions, `valF` and `valD`, mapping function and data object names to values:

record `chip_data =`
```
  valF :: "fn ⇀ val"
  valD :: "dn ⇀ val"
```

The function `val` takes an argument of type `chip_data` and yields a partial function lifting `valF` and `valD` to general object names of type `on`:

constdefs
```
  val :: "chip_data ⇒ on ⇀ val"
 "val s on ≡ case on of F fn ⇒ valF s fn  | D dn ⇒ valD s dn"
```

Having defined the two components of the processor state, we can now give the definition of the overall state:

```
types SLE66_state = "ph × chip_data"
```

We will often need to refer to the set of functions available in the current state, therefore we introduce an auxiliary function *fct* that yields the domain of *valF*:

```
constdefs
  fct :: "chip_data ⇒ fn set"
 "fct s ≡ dom (valF s)"
```

We declare three further auxiliary functions that denote the results and state changes of a processor function (including test functions):

```
consts
 "output"   :: "fn ⇒ chip_data ⇒ val"
 "change"   :: "fn ⇒ chip_data ⇒ chip_data"
                                  — change is unused for test functions
 "positive" :: "val ⇒ bool"  — check for positive test outcome
```

Further ISM section ingredients We need only two port names, one for input to the chip and one for its output:

```
datatype interface = In | Out
```

SLE 66 commands provide information on the subjects issuing them. There is a special subject *Pmf* denoting the processor manufacturer.

```
typedecl sb
consts Pmf :: sb
```

Possible input consists of either the two kinds of SLE 66 commands modeling function execution and function code loading operations or the *Spy* operation, which models attacks that may reveal information stored on the chip and may corrupt the chip memories. Output of the SLE 66 may be the result value of a (regular) function or an indication of success or failure.

```
datatype message =
  Exec sb fn | Load sb fn val | Spy on  — input
| Val val | Ok | No                     — output
```

The subjects performing regular commands identify themselves to the chip via physical means. The actual authentication mechanism, as well as many other implementation details, is confidential and beyond the scope of this article anyway. Here we just declare an auxiliary function that yields the subject issuing a (regular) command:

```
consts subject :: "message ⇒ sb"
primrec
  "subject (Exec sb fn  ) = sb"
  "subject (Load sb fn v) = sb"
```

ISM definition Having defined its various parameters, we can finally give the theory section that specifies the SLE 66 model as an ISM:

ism *SLE66* =
  ports *interface*
    inputs *"{In}"*
    outputs *"{Out}"*
  messages *message*
  states
    control *ph* **init** *"P0"*
    data *chip_data* **name** *"s"* — The data state variable is called *s*. Note that the initial data state is left unspecified and thus is arbitrary, which is a good example of underspecification since its actual value is immaterial for the security properties we are interested in.

  transitions

— Rule R00 specifies execution of a test function *f* from the set *FTest0* by the processor manufacturer *Pmf* in the initial phase *P0*. If the test is successful then the SLE 66 enters the next phase *P1*, answers with *Ok*, and disables the test functions *FTest0*. As specified by the data theory subsection just above, the variable *s* denotes the current data state of the ISM at the beginning of the transition. Thus, for example, *fct s* means the functions currently available. The operator '⌊' below restricts a partial function, in this case *valF s*, to the given set, in this case the complement of *FTest0*.
Rule R00 is typical for interactions of the SLE 66 in the sense that a single input triggers a single output. Note that the direct relation of input and output is expressed easily using ISMs, whereas using IOAs, two transitions would be required whose relation would be cumbersome to express and to use during verification.

  R00: P0 → P1
    pre *"f ∈ fct s∩FTest0", "positive (output f s)"*
    in *In "[Exec Pmf f]"*
    out *Out "[Ok]"*
    post *valF := "valF s⌊(-FTest0)"*

— Rule R01 is analogous to R00, but specifies that if the test function *f* is from *FTest1* rather than *FTest0* then phase *P1* is skipped and the chip enters *P2* immediately, disabling all test functions *FTest*:

  R01: P0 → P2
    pre *"f ∈ fct s∩FTest1", "positive (output f s)"*
    in *In "[Exec Pmf f]"*
    out *Out "[Ok]"*
    post *valF := "valF s⌊(-FTest)"*

— If in *P0* a test function gives a negative result then the chip enters the *Error* phase and the output is *No*:

  R02: P0 → Error
    pre *"f ∈ fct s∩FTest0", "¬positive (output f s)"*
    in *In "[Exec Pmf f]"*
    out *Out "[No]"*

— Any other function call issued by the processor manufacturer in *P0* has the standard consequences: The function result is output and the data state changed (according to the semantics of the function which is not further specfied). Note that by the form of

postcondition used, the whole data state (consisting of *valF* and *valD* here) is replaced by the given value: the denotation of *change f s*.

R03: P0 → P0
  pre *"f ∈ fct s - FTest"*
  in *In "[Exec Pmf f]"*
  out *Out "[Val (output f s)]"*
  post *"change f s"*

— In all remaining cases for phase 0, the attempted function execution is ignored and the output is *No*:

R04: P0 → P0
  pre *"sb ≠ Pmf ∨ f ∉ fct s"*
  in *In "[Exec sb f]"*
  out *Out "[No]"*

— This ends the specifications of transitions originating in *P0*.

The specifications of transitions originating in *P1* are fully analogous to the rules R00, R02, R03, and R04, just replacing *P0* by *P1*, *P1* by *P2*, and *FTest0* by *FTest1*:

R11: P1 → P2
  pre *"f ∈ fct s∩FTest1", "positive (output f s)"*
  in *In "[Exec Pmf f]"*
  out *Out "[Ok]"*
  post *valF := "valF s⌊(-FTest1)"*

R12: P1 → Error
  pre *"f ∈ fct s∩FTest1", "¬positive (output f s)"*
  in *In "[Exec Pmf f]"*
  out *Out "[No]"*

R13: P1 → P1
  pre *"f ∈ fct s - FTest1"*
  in *In "[Exec Pmf f]"*
  out *Out "[Val (output f s)]"*
  post *"change f s"*

R14: P1 → P1
  pre *"sb ≠ Pmf ∨ f ∉ fct s"*
  in *In "[Exec sb f]"*
  out *Out "[No]"*

— The rules R21 and R22 specify function calls in *P2* analogously to R03 and R04, except that any subject is allowed to issue them:

R21: P2 → P2
  pre *"f ∈ fct s"*
  in *In "[Exec sb f]"*
  out *Out "[Val (output f s)]"*
  post *"change f s"*

R22: P2 → P2
  pre *"f ∉ fct s"*
  in *In "[Exec sb f]"*
  out *Out "[No]"*

— In the *Error* phase, the only function that may be called is chip identification, yielding the serial number *SN*. All other cases yield *No*:

```
R31: Error → Error
  pre "f_SN ∈ fct s"
  in  In  "[Exec sb f_SN]"
  out Out "[Val SN]"
R32: Error → Error
  pre "f ∉ fct s∩{f_SN}"
  in  In  "[Exec sb f]"
  out Out "[No]"
```

— The rules R41 and R42 specify the behavior of the *Load* operation, which is allowed only for the processor manufacturer and only in the upload phase *P1*. If allowed, *valF* is updated at the position *f* with the new function value *v*.
In contrast to the original LKW model [LKW00], the *Load* operation may upload not only non-security-relevant functions but also functions of the application security domain (as long as no such function of the same name is already present).

```
R41: P1 → P1
  pre "f ∈ F_NSec ∪ (F_ASec - fct s)"
  in  In  "[Load Pmf f v]"
  out Out "[Ok]"
  post valF := "valF s(f↦v)"
R42: ph → ph
  pre "f ∉ F_NSec ∪ (F_ASec - fct s) ∨ sb ≠ Pmf ∨ ph ≠ P1"
  in  In  "[Load sb f v]"
  out Out "[No]"
```

— Note that the rule R42 is generic in the sense that it applies to more than one control state of the ISM, namely all phases except *P1*.

— The rules R51 thru R53 specify the possible reactions of the chip to attacks, modeled by the *Spy* operation. If the attacker attempts to read a non-secret object whose name is *on* and the chip is not in the *Error* phase, the access may be granted, yielding the desired value (if any):

```
R51: ph → ph
  pre "on ∉ Sec", "ph ≠ Error"
  in  In  "[Spy on]"
  out Out "case val s on of None ⇒ [] | Some v ⇒ [Val v]"
```

— Rule R52 specifies the typical reaction of the SLE 66 upon attacks trying to read a secret object while tampering with the chip: it may be unable to prevent that the desired value is output, but in any case it reaches the *Error* phase from which no further secrets may be obtained, as specified by the rules R31, R32, and R53.

```
R52: ph → Error
  pre "on ∈ Sec", "v ∈ {[],[Val (the (val s on))]}", "ph ≠ Error"
  in  In  "[Spy on]"
  out Out "v"
  post "any"
```

— Note that R52 describes two sorts of nondeterminism: *v* denotes either the empty output or the singleton output giving the desired value, and the attack may corrupt the function and data stores arbitrarily.

There are also cases where the chip can resist an attack without any damage and without any leakage of secrets, such that there is no need to enter the *Error* phase:

```
R52':ph → ph
  pre "on ∈ Sec", "ph ≠ Error"
  in  In  "[Spy on]"
  out Out "[]"
```

— If the chip is already in the *Error* phase, no further secrets can be obtained. The chip state may be corrupted further, but it makes sure that it stays locked in the *Error* phase:

```
R53: Error → Error
  in  In  "[Spy on]"
  out Out "[]"
  post "any"
```

As expressed by the rules R52 and R53, the attacker may obtain (the representation of) at most one secret object from the chip memory. It is interesting to observe that the leakage of one item is harmless because all data stored on the chip is encrypted. There are two cases to consider:

- The secret obtained is the de-/encryption key itself, which is not helpful to the attacker because no further data item, in particular none encrypted with the key, can be obtained.
- The secret obtained is an encrypted value, which is not helpful because the attacker cannot any more obtain the decryption key.

Obviously, sophisticated techniques are required to implement the specified reaction to physical attacks modeled by the *Spy* operation.

ISM runs The SLE 66 ISM just defined models the static interface of the chip as well as all possible single state transitions that it can perform. In order to describe the overall behavior of the chip during its life-cycle, we can refer to the notions that our Isabelle implementation provides for ISMs in general:

```
types
  SLE66_trans = "(unit, interface, message, SLE66_state) trans"
constdefs
  Trans :: "SLE66_trans set"          — the set of all possible transitions
 "Trans ≡ trans SLE66.ism"

  TRuns :: "(SLE66_trans list) set"  — all possible transition sequences
 "TRuns ≡ truns SLE66.ism"

  Runs :: "(SLE66_state list) set"   — all possible sequences of states
 "Runs ≡ runs SLE66.ism"
```

This concludes the system model of the SLE 66.

4.5 Properties

The second part of the SLE 66 security model deals with the security properties derivable from the system model.

Security Objectives In the (confidential[8]) original security requirements specification by Infineon, the security objectives for the SLE 66 had been stated as follows.

SO1. "The hardware must be protected against unauthorised disclosure of security enforcing functionality."
SO2. "The hardware must be protected against unauthorised modification of security enforcing functions."
SO3. "The information stored in the processor's memory components must be protected against unauthorised access."
SO4. "The information stored in the processor's memory components must be protected against unauthorised modification."
SO5. "It may not occur that test functions are executed in an unauthorised way."

Later, an additional requirement concerning the confidentialiy and integrity of Smartcard Embedded Software, which is not part of the security enforcing functionality, has been added [AHIP01, §4.1].

Having formally defined the SLE 66 system model, these informal statements can now be expressed formally as predicates on the system behavior, describing unambiguously and in detail which states may be reached under which circumstances, which data may be modified, and which output may appear on the output channel.

After formalizing the security objectives, it is natural to ask if the chip behavior, as specified in the system model, actually fulfills these requirements. The corresponding proofs have been conducted first using pen and paper, as reported in [LKW00]. Within the ISM framework, we meanwhile have verified these properties even mechanically using Isabelle, discovering two major flaws that will be reported in this subsection. Below we give all the required auxiliary definitions, the most important lemmata, and all theorems, together with an abstract informal description of the machine-checked proofs.

Model Assumptions Due to the abstract specification style where e.g. the semantics of parts of the chip functionality is not fully specified, it turns out that in order to prove the properties, a few general axioms that augment the system model are required. The first one of them asserts that security-relevant functions do not modify security-relevant functions:

```
Axiom1: "f∈fct s∩F_Sec ⟹ valF (change f s)⌊F_Sec = valF s⌊F_Sec"
```

[8] quotations with permission.

In comparison to the version of this axiom in the original model, the scope of functions `f` has been extended from "initially available" to "security-relevant", reflecting the changes to rule `R41`. Part of the lemmas as well as the formalized security objective `FSO21` change accordingly.

The second axiom is very similar, stating that also non-security-relevant functions do not modify security-relevant functions:

```
Axiom2: "f∈fct s∩F_NSec ⟹ valF (change f s)⌊F_Sec = valF s⌊F_Sec"
```

In order to formalize the security objective SO1 and `Axiom3`, we define the set `ValF_Sec r` holding all code of security-relevant functions in a given run (i.e., sequence of states) `r`.

constdefs

```
  ValF_Sec :: "SLE66_state list  ⇒ val set"
 "ValF_Sec r ≡ ⋃{ran (valF s⌊F_Sec) |ph s. (ph,s) ∈ set r}"
```

The third (and last) axiom introduced in the LKW model states that in phase 2, a function cannot reveal (by intentional "guessing" or by accident) any members of `ValF_Sec r`. This rather self-evident requirement is needed for technical reasons in the proof of SO1.

```
Axiom3: "⟦r∈Runs; (P2,s)∈set r; f∈fct s⟧⟹ output f s∉ValF_Sec r"
```

A notational remark is in order here: in Isabelle formualas, multiple premises are bracketed using '⟦' and '⟧' and separated using '`;`'.

When machine-checking the proofs contained in [LKW00] with Isabelle, we noticed that a fourth axiom was missing that makes an implicit but important assumption explicit: if a function object may be referenced in two (different) ways and one of them declares the function to be security-relevant, the other has to do the same.

```
  Axiom4: "⟦r ∈ Runs;
            (ph, s) ∈ set r; (ph', s') ∈ set r;
            val s n = Some v; val s' n' = Some v;
            n ∈ Sec⟧ ⟹ n' ∈ Sec"
```

Such experience of missing cricial assumptions demonstrates how important machine support is when conducting formal analysis.

Theorems Finally, we translate the five informal security objectives to Isabelle formulas and prove them within the system. It is instructive to compare the formal versions of the security objectives FSOx below with the informal ones, SOx, given above.

The formalization of SO1, called `FSO1`, states that in any sequence `ts` of transitions performed by the chip, if the chip outputs any value `v` representing the code of any security-relevant function during its hitherto life, then the error state is entered or the output was in response to a function execution request by the processor manufacturer:

theorem *FSO1:* "⟦*ts* ∈ *TRuns*; ((*p*,(*ph*,*s*)),*c*,(*p'*,(*ph'*,*s'*))) ∈ *set ts*; *p' Out* = [*Val v*]; *v* ∈ *ValF_Sec* (*truns2runs ts*)⟧ ⟹ *ph'* = *Error* ∨ (∃*fn*. *p In* = [*Exec Pmf fn*])"

The proof of FSO1 proceeds by unfolding some definitions, e.g. of the SLE 66 ISM, applying properties of auxiliary concepts like *truns2runs*, and a case split on all possible transitions. Isabelle can solve most of the cases automatically (with straightforward term rewriting and purely predicate-logical reasoning), except for two: the case of rule R21 is handled using Axiom3, and for R51 we rely on the property "⟦*r* ∈ *Runs*; (*ph*, *s*) ∈ *set r*; *v* ∈ *ValF_Sec r*; *val s n* = *Some v*⟧ ⟹ *n* ∈ *Sec*" which in turn relies on Axiom4.

A more elaborate formalization of SO1 and SO3 taking into account also indirect and partial information flow is motivated and sketched in [Ohe04].

Like in the original LKW model, the translation of SO2 splits into two parts. FSO21' states that for any (even unreachable) transition not ending in the error phase, if a security-relevant function *g* is present in both the pre-state and the post-state, the code associated with it stays the same:

theorem *FSO21':* "⟦((*p*,(*ph*,*s*)),*c*,(*p'*,(*ph'*,*s'*)))∈*Trans*; *ph'* ≠ *Error*; *g* ∈ *fct s*∩*fct s'*∩*F_Sec*⟧ ⟹ *valF s' g* = *valF s g*"

This property is a generalization of the original FSO21, reflecting the extensions made to the *Load* operation in rule R41: Here we do not compare the initial and current value of *g* but the previous and current one, which takes into account also functions added in the meantime.

The proof of this property is — as usual — by case distinction over all possible transitions. Most cases are trivial except for those where function execution may change the stored objects, which are described by the rules R03, R13, and R21. Here an argumentation about the invariance of security-relevant functions *g* is needed, which follows easily from Axiom1 and Axiom2.

Similarly to FSO21', FSO22 states that for any transition within the same phase that is not the error phase, the set of existing security-relevant functions is non-decreasing:

theorem *FSO22:* "⟦((*p*,(*ph*,*s*)),*c*,(*p'*,(*ph'*,*s'*)))∈*Trans*; *ph'* ≠ *Error*; *ph* = *ph'*⟧ ⟹ *fct s*∩*F_Sec* ⊆ *fct s'*∩*F_Sec*"

Not surprisingly, the proof of this property is completely analougous.

FSO3 states that if the attacker obtains a result trying to get hold of a security-relevant data object *on*, then the chip enters the error phase:

theorem *FSO3:*"⟦((*p*,(*ph*,*s*)),*c*,(*p'*,(*ph'*,*s'*)))∈*Trans*; *p In* = [*Spy on*]; *on* ∈ *Sec*; *p' Out* ≠ []⟧ ⟹ *ph'* = *Error*"

The proof is done simply by case distinction.

FSO4 states that any transition not entering the error phase but changing the state does this in a well-behaved way: *s'* is derived from *s* via the desired effect of executing an existing function, or there is a phase change where only the test functions may be modified, or only a single function *f* is changed due to a *Load* operation:

theorem *FSO4:*
```
"⟦((p,(ph,s)),c,(p',(ph',s'))) ∈ Trans; ph' ≠ Error⟧ ⟹
 s' = s ∨
 (∃sb f  . p In = [Exec sb f  ] ∧ f ∈ fct s ∧ s' = change f s) ∨
 (ph'≠ph ∧ valD s' = valD s ∧ valF s'⌊(-FTest) = valF s⌊(-FTest))∨
 (∃sb f v. p In = [Load sb f v] ∧
           valD s' = valD s ∧ valF s'⌊(-{f}  ) = valF s⌊(-{f}  ))"
```

The proof is also straightforward by case distinction.

A second omission of the LKW model was that in the proof of the security objective FSO5 an argumentation about the accessibility of certain functions was not given in a rigorous way. We fix this by introducing an auxiliary property (where, as typical with invariants, finding the appropriate one is the main challenge) and proving it to be an invariant of the ISM. The invariant states that in phase 1, the test functions from *FTest0* have been disabled, and in phase 2, all test functions have been disabled:

constdefs
```
  no_FTest_invariant :: "SLE66_state ⇒ bool"
 "no_FTest_invariant ≡ λ(ph,s).
  ∀f ∈ fct s. (ph = P1 ⟶ f ∉ FTest0) ∧ (ph = P2 ⟶ f ∉ FTest)"
```

When proving that the invariant holds, 14 of the 19 cases are trivial, and the remaining ones require simple properties of the set *FTest*, and two of them require additionally Axiom1 and Axiom2. The invariant implies

lemma *P2_no_FTest:*
```
  "⟦(P2,s) ∈ reach SLE66.ism; f ∈ fct s⟧ ⟹ f ∉ FTest"
```

Wxploiting this property for the case of rule R21, we can prove FSO5 in the usual way. This theorem states that in any sequence of transitions performed by the chip, any attempt to execute a test function not issued by the processor manufacturer is refused:

theorem *FSO5:*
```
"⟦ts ∈ TRuns; ((p,(ph,s)),c,(p',(ph',s'))) ∈ set ts;
  p In = [Exec sb f]; f ∈ FTest⟧ ⟹
  sb = Pmf ∨ s' = s ∧ p' Out = [No]"
```

The Isabelle proofs of all six theorems formalizing the security objectives and the two lemmas required are well supported by Isabelle: each of them takes just a few steps, about half of which are automatic.

end

This finishes our detailed presentation of the SLE 66 case study. It demonstrates that the ISM approach can be fruitfully applied to both model and prove the security properties of state transition systems. The use of mechanical type checks and theorem proving system ensures a level of accuracy hardly reachable in a pen-and-paper analysis.

5 Needham-Schroeder Public-Key Protocol

In contrast to the high-level requirements analysis of the rather state-oriented SLE 66 model described in the previous section, we now turn to a more low-level analysis of a communication-oriented system. Our aim is to demonstrate that the ISM approach is capable of handling such quite different systems in a both rigorous and elegant way as well.

As a typical example for such a distributed system, we take Lowe's fix of the *Needham-Schroeder public-key authentication protocol* [Low96], which we call *NSL*. The emphasis here is not to provide new insights to the protocol, but to use a well-known benchmark system that makes our approach easy to compare with many other approaches that have been used to model (essentially) the same system.

We base our ISM model on the formalization by Paulson [Pau98]. His so-called "inductive approach" is tailored to semi-automated verification of cryptographic protocols. Its great advantage is a high degree of automation, due to abstraction to the core semantics of the protocols: event traces. On the other hand, this makes both the models and the properties at least cumbersome to express: state information is implicit, yet often it has to be referred to, which is done by repeating suitable parts of the event history and sometimes even by introducing auxiliary events.

5.1 AutoFocus Diagrams

As usual, our model of the NSL system consists of an agent called `Alice` aiming to establish an authenticated session with another agent called `Bob` in the presence of an `Intruder` according to the Dolev-Yao attacker model [DY83]. As

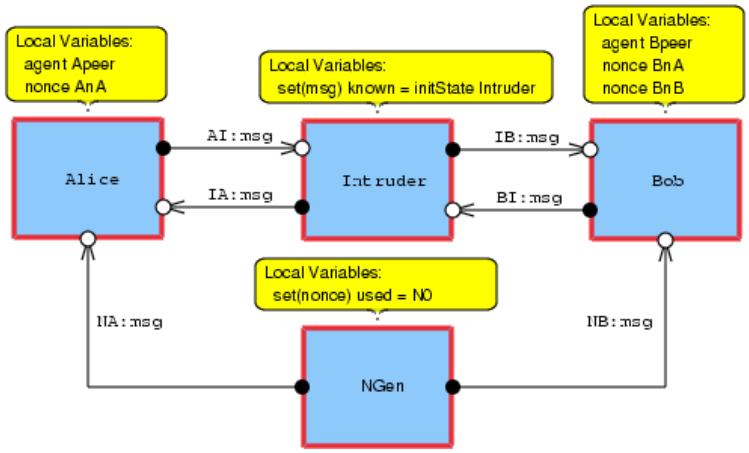

Fig. 6. NSL System Structure Diagram

will be motivated in §5.2, we furthermore introduce a server ISM called `NGen` that generates nonces for all honest agents. The corresponding system structure diagram in Figure 6 shows the four components with their data state (reflecting the expectations of the two agents, the set of messages the intruder knows of, and the set of already used nonces, respectively) and the named connections between them.

Even if sometimes neglected, agents involved in communication protocols do have state: their current expectations and knowledge. This is made explicit in a convenient way by describing their interaction behavior with state transition diagrams. Figure 7 shows the three states of the agent `Alice` and the transitions between them, which have the general format `guard : inputs : outputs : assignments`.

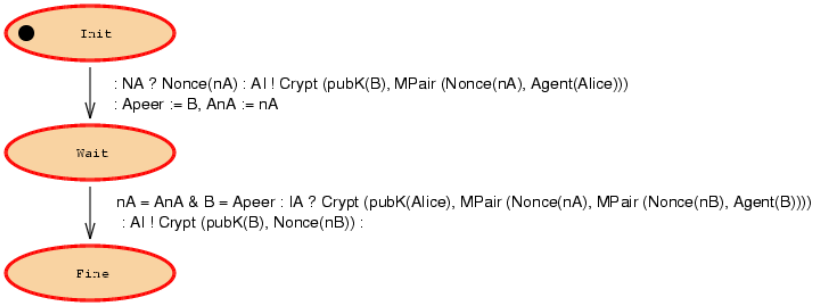

Fig. 7. NSL State Transition Diagram: `Alice`

In the initial state, `Alice` decides which agent `B` she wants to talk to and sends the corresponding request consisting of a fresh nonce `nA` (which she has obtained from the nonce server via her port `NA`) and her identity `Alice`, encrypted under the public key of the intended receiver. This message is actually sent to the port `AI` of the intruder. Alice remembers her intended peer in the local variable `Apeer` and the nonce she has used in the variable `AnA`. In the next state she awaits a response from the prospective peer, decrypts it and checks its authenticity by comparing the nonce value `nA` and agent name `B` with the corresponding items in her memory. Only if the decryption and the two comparisons are successful, the transition to her final state actually takes place, sending an appropriate acknowledgment to her peer and storing the nonce `nB` just received in her variable `AnB`. The third state represents (hopefully) successful session establishment where all essential parameters of the session may be referred to by the local variables of `Alice`.

From the example of `Alice`'s transitions, one realizes that ISM control state information is a natural way of fixing the order of protocol steps.

`Bob`'s state transitions are analogous and thus not shown here.

The transitions of the `Intruder` are quite different from the regular protocol participants: it stores all messages received on its ports `AI` and `BI` in the local variable *known* and can send any message derivable from its current knowledge (by analyzing the messages contained in the set *known*, utilizing the decryption keys it knows of, and synthesizing messages from the resulting pieces) to the ports `IA` and `IB`, as depicted by Figure 8. The figure reveals a weakness of

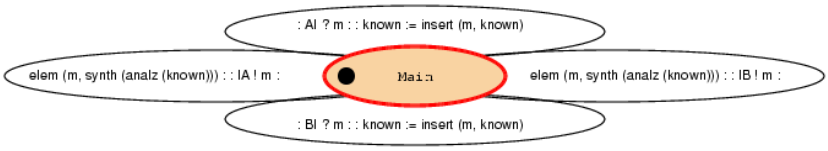

Fig. 8. NSL State Transition Diagram: `Intruder`

modeling with AutoFocus: there is a lot of redundancy among each of the two pairs of transitions (where the difference is just in the port names used), which can be avoided in the Isabelle representation by using generic transitions (where the port used for input or output is a variable that may hold either of the two possible values, as shown below).

Note that the intruder may take part in any number of sessions simultaneously. If the analysis needs to include the possibility that a regular agent takes part in more than one protocol run simultaneously, this can be modeled by multiple instantiation of the respective agent — under the assumption that from that agent's perspective the protocol runs are independent of each other.

The transition diagram of `NGen` is similar to the one of the intruder, except that there are no transitions with input.

5.2 Isabelle Definition

This section gives parts of our Isabelle representation of NSL. Refer to §3.4 for the details of ISM sections. We do not show the definitions of the various state and message components here since they are straightforward and analogous to the SLE66 model. Moreover, we give only the ISM definitions of those components for which we have not already given an AutoFocus STD above.

ism `Bob` =
  ports *channel*
    inputs *"{NB,IB}"*
    outputs *"{BI}"*
    messages *msg*
  states *state* — this is the sum of the four state types of the system components, required because of the type problem mentioned in §2.1
    control *B_control* **init** *"Idle"*
    data *B_data* **init** *"B0"*

```
  transitions
Resp: Idle → Resp
     in   NB "[Nonce nB]",
          IB "[Crypt (pubK Bob) {|Nonce nA, Agent A|}]"
     out  BI "[Crypt (pubK A) {|Nonce nA, Nonce nB, Agent Bob|}]"
     post "Bpeer := A, BnA := nA, BnB := nB"
Ack': Resp → Conn
     pre  "nB = BnB s"
     in    IB "[Crypt (pubK Bob) (Nonce nB)]"
```

Note that `Bob`'s first transition `Resp` takes two inputs, from the nonce generator and the intruder, and produces one output. If we modeled this transition using IOAs, we would have needed three transitions with intermediate states. The precondition of transition `Ack'` could have been made implicit by moving the comparison as a pattern to the **in** part, yet we make it explicit in order to emphasize its importance. The local variable `BnB` is used to store the value of the nonce expected, while the other two variables express `Bob`'s view to whom he is connected in which session. In Paulson's model, this state information is implicit in the event trace.

Modeling the freshness of nonces is an interesting problem, for which we are aware of essentially four solutions, each with their pros and cons.

- In Paulson's model [Pau98], nonces are generated non-deterministically under the side condition that they do not already appear in the current message/event history. This criterion refers to the semantic and system-global notion of event traces — something not available from the (local) perspective of ISMs.
- One could combine local and global freshness conditions and let each agent generate its own nonces: by producing fresh values locally and combining them with the globally unique agent identifier. The drawback of this solution is that each nonce issuer has to implement the mechanism just described.
- One could enforce global freshness by adding an axiom restricting system runs in the desired way. We prefer a more constructive approach here and derive the required freshness property as a lemma.
- Our solution is to introduce a nonce server component called `NGen` that performs the generation of nonces for all agents in a centralized fashion. In this way we can ensure global freshness with a constructive local criterion.

A further motivation to us for selecting the fourth solution just mentioned was that it makes the communication patterns of the agents more interesting because `Bob` has a transitions that inputs from two sources simultaneously. Note that `NGen` is just a modeling aid and thus its correct interplay with the agents, including authentication issues, does not need to be analyzed.

The ISM definition of `NGen` is rather simple because `NGen` does not require control state information and its local state consists only of the single variable storing the set of all nonces that already have been used. Therefore, we may identify the whole local state with this variable and call it *used*, eliminating

the need to define a record type and use the corresponding record selectors and updates.

```
ism NGen =
   ports channel
     inputs    "{}"
     outputs   "{NA,NB}"
     messages  msg
   states state
     data "nonce set" init "N0" name "used"
   transitions
 Cackle:
      pre   "ch ∈ {NA, NB}", "n ∉ used"
      out    ch "[Nonce n]"
      post "insert n (used)"
```

Note that the output port `ch` is (non-deterministically) selected from the set of two distinct names, which ensures the exclusive use of each nonce.

The family of all four ISMs is composed in parallel to form the NSL system. It is easy to prove that this ISM family is closed and all its members, as well as their parallel composition, are well-formed.

5.3 Properties

Properties of protocols specified with ISMs may be expressed with reference to both the state of agents and the messages exchanged. In the case of NSL, the most interesting property is authentication of `Alice` to `Bob` (actually, even session agreement [Low97] from `Bob`'s view), which we formulate as

```
⟦Alice ∉ bad;  Bob ∉ bad; (b,s)#cs ∈ NSL_Runs⟧ ⟹
 (∃nA. Bob_state   s  = (Conn, (|Bpeer=Alice, BnA=nA, BnB=nB|))) ⟶
 (∃(b',s') ∈ set cs.
 (∃nA. Alice_state s' = (Wait, (|Apeer=Bob  , AnA=nA, AnB=nB|))))
```

This can be quite intuitively read as: if in the current state `s` of the system `Bob` believes to be connected to `Alice` within a session identified by the nonce `nB` then there is an earlier state `s'` where `Alice` was in the waiting state referring to the same nonce `nB` after initiating a connection with `Bob`.

It is interesting to compare the above formulation with Paulson's[9]:

```
⟦A ∉ bad;  B ∉ bad;  evs ∈ ns_public;
 Crypt (pubK B) (Nonce NB) ∈ parts (spies evs);
 Says B A (Crypt (pubK A) {|Nonce NA,Nonce NB,Agent B|}) ∈ set evs
⟧ ⟹
 Says A B (Crypt (pubK B) {|Nonce NA,Agent A|}) ∈ set evs
```

[9] http://isabelle.in.tum.de/library/HOL/Auth/NS_Public.html

This statement is necessarily more indirect than ours since the beliefs of the agents have to be coded by elements of the event history. At least in this case, all messages of a protocol run have to be referred to. Note further that this formulation makes stronger assumptions than ours because an agreement on the value of the nonce `NB` is involved.

Due to the extra detail concerning agent state and the input buffers (which are not actually required for the NSL protocol), the proofs within the ISM approach are more painful and require more lemmas about intermediate states of protocol runs than Paulson's inductive proofs. On the other hand, the semi-automatic proofs within the ISM approach probably scale better.

There are about a dozen lemmas proved by rule induction, most of which deal with the freshness and usage of nonces generated by `NGen`. The main theorem is proved employing a variant of Schneider's rank function approach [Sch97], which we describe in detail in [Ohe02, §3].

6 Conclusion

ISMs are designed as high-level I/O automata, with additional structure and communication facilities. Like IOAs, ISMs are suitable for describing typical state-based communication systems relevant for security analysis, where ISM provide increased simplicity wrt. specifying component interaction via buffered communication and means to directly relate input and output actions within a single transition.

We have shown that the ISM approach is equally applicable to a variety of security analysis tasks, ranging from high-level security modeling and requirements analysis, typically showing less system structure but increased complexity of state transitions, to security analysis of distributed systems including cryptographic protocols, likely to exhibit advanced system structuring. The examples explicate the importance of a fully formalized strategy and mechanized proofs. In particular, the LKW model has been significantly improved by identifying hidden assumptions and completing sloppy argumentation.

The ISM approach offers graphic representation by means of AutoFocus System Structure Diagrams and State Transitions Diagrams. A tool program closely relates these graphical development and documentation capabilities with the formal system specification and verification capabilities of the mechanical theorem prover Isabelle/HOL.

Further work on ISMs includes the extension of the proof support in the ISM level concerning e.g. refinement and the provision of a specification language based on temporal logic. Additional AutoFocus capabilities may be made available, including further systems views like event traces and simulation, as well as test case generation.

In brief, the Interacting State Machines approach turns out to offer good support for formal security analysis in the way required within an industrial environment, meeting the goals stated in §1.2.

Acknowledgements. We thank Guido Wimmel, Thomas Kuhn and several anonymous referees for their comments on earlier versions of this article.

References

AGKS99. David Aspinall, Healfdene Goguen, Thomas Kleymann, and Dilip Sequeira. *Proof General*, 1999. http://www.proofgeneral.org/.

AHIP01. Atmel, Hitachi Europe, Infineon Technologies, and Philips Semiconductors. Smartcard IC Platform Protection Profile, Version 1.0, July 2001. http://www.bsi.de/cc/pplist/ssvgpp01.pdf.

But99. Michael Butler. csp2B : A practical approach to combining CSP and B. In *Proc. of FM'99: World Congress on Formal Methods*, pages 490–508, 1999.

CC99. Common Criteria for Information Technology Security Evaluation (CC), Version 2.1, 1999. ISO/IEC 15408.

DY83. Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.

Fis00. Clemens Fischer. *Combination and implementation of processes and data: from CSP-OZ to Java*. PhD thesis, Univ. of Oldenburg, 2000.

GL98. Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, MIT, August 1998.

HSSS96. Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus - a tool for distributed systems specification. In *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *LNCS*, pages 467–470. Springer-Verlag, 1996. See also http://autofocus.in.tum.de/index-e.html.

ITS91. Information Technology Security Evaluation Criteria (ITSEC), 1991.

JW01. Jan Jürjens and Guido Wimmel. Formally testing fail-safety of electronic purse protocols. In *Automated Software Engineering*. IEEE Computer Society, 2001.

Kay01. Dilsun Kirli Kaynar. IOA language and toolset, 2001. http://theory.lcs.mit.edu/tds/ioa.html.

KO03. Thomas Kuhn and David von Oheimb. Interacting State Machines for mobility. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proc. of the* 12^{th} *International FME Symposium (FM'03)*, volume 2805 of *LNCS*. Springer, September 2003. http://ddvo.net/papers/ISMfM.html.

LKW00. Volkmar Lotz, Volker Kessler, and Georg Walter. A Formal Security Model for Microprocessor Hardware. In *IEEE Transactions on Software Engineering*, volume 26, pages 702–712, August 2000.

Low96. Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. of TACAS*, volume 1055 of *LNCS*, pages 147–166. Springer-Verlag, 1996.

Low97. Gavin Lowe. A hierarchy of authentication specifications. In *10th Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 1997.

LT89. Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989. http://theory.lcs.mit.edu/tds/papers/Lynch/CWI89.html.

Mül98. Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Technische Univerität München, 1998. See also http://isabelle.in.tum.de/IOA/.

Nan02. Sebastian Nanz. Integration of CASE tools and theorem provers: a framework for system modeling and verification with AutoFocus and Isabelle. Master's thesis, TU München, 2002. http://www.doc.ic.ac.uk/~nanz/publications/csthesis/.

NPW02. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. See also http://isabelle.in.tum.de/docs.html.

Ohe02. David von Oheimb. Interacting State Machines: *a stateful approach to proving security*. In Ali E. Abdallah, Peter Ryan, and Steve Schneider, editors, *Formal Aspects of Security*, volume 2629 of *LNCS*, pages 15–32. Springer-Verlag, 2002. http://ddvo.net/papers/ISMs.html.

Ohe04. David von Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In P. Samarati, P. Ryan, D. Gollmann, and R. Molva, editors, *Computer Security – ESORICS 2004*, volume 3193 of *LNCS*, pages 225–243. Springer, 2004. http://ddvo.net/papers/Noninfluence.html.

OL03. David von Oheimb and Volkmar Lotz. Generic Interacting State Machines and their instantiation with dynamic features. In Jin Song Dong and Jim Woodcock, editors, *Formal Methods and Software Engineering (ICFEM)*, volume 2885 of *LNCS*, pages 144–166. Springer, November 2003. http://ddvo.net/papers/GenISMs.html.

OLW02. David von Oheimb, Volkmar Lotz, and Gerog Walter. An interpretation of the LKW model according to the SLE66CX322P security target. Unpublished, January 2002.

OLW04. David von Oheimb, Volkmar Lotz, and Georg Walter. Analyzing SLE 88 memory management security using Interacting State Machines. *International Journal of Information Security*, 2004. To appear; preprint: http://ddvo.net/papers/SLE88_MM.html.

ON02. David von Oheimb and Sebastian Nanz. *ISM Homepage: Documentation, sources and distribution*, 2002. http://ddvo.net/ISM/.

OWL03. David von Oheimb, Georg Walter, and Volkmar Lotz. A formal security model of the infineon SLE 88 smart card memory management. In *Proc. of the 8th European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *LNCS*. Springer, 2003. http://ddvo.net/papers/SLE88_MM.html.

Pau98. Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

PNW+. Lawrence Paulson, Tobias Nipkow, Markus Wenzel, et al. The Isabelle/HOL library. http://isabelle.in.tum.de/library/HOL/.

S+. Oscar Slotosch et al. Validas Model Validation AG. http://www.validas.de/.

Sch97. Steve Schneider. Verifying authentication protocols with CSP. In *10th Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, June 1997.

WW01. Guido Wimmel and Alexander Wisspeintner. Extended description techniques for security engineering. In M. Dupuy and P. Paradinas, editors, *International Conference on Information Security (IFIP/SEC)*. Kluwer Academic Publishers, 2001. http://www4.in.tum.de/papers/WW01.html.

Index