Interacting State Machines
– a stateful approach to proving security –

David von Oheimb

Siemens AG, Corporate Technology, D-81730 Munich
David.von.Oheimb@siemens.com

Abstract. We introduce Interacting State Machines (ISMs), a general formalism for abstract modeling and verification of reactive systems. We motivate and explain the concept of ISMs and describe their graphical representation with the CASE tool AutoFocus. The semantics of ISMs is defined using Higher-Order Logic within the theorem prover Isabelle. ISMs can be seen as high-level variants of Input/Output Automata, therefore we give also a semantic translation from ISMs to IOAs.
By the "benchmark" example of Lowe's fix of the Needham-Schroeder protocol we demonstrate the strengths of the ISM approach to express and prove security properties in a both elegant and machine-checked way.

1 Motivation and Related Work

When investigating the correctness, safety and security of complex IT systems, formal modeling and verification of their key properties are essential for fulfilling strong quality requirements. A prominent example of this approach is security analysis according to the upper evaluation assurance levels defined in the IT security evaluation criteria catalog ITSEC [ITS91] and its successor, the Common Criteria [CC99].

We present a formalism and tool support facilitating abstract formal analysis of a wide range of reactive IT systems including smart cards, embedded systems, network protocols, operating systems, databases, etc. During the development of the framework [ON02], an important requirement was that it must be simple and practical enough for industrial application where results are to be obtained quickly and with limited effort. Therefore it should allow to express the key aspects of such systems in a convenient, flexible, and intuitive way. It should be supported by a well-developed theory and mature tools for verification and textual as well as graphical documentation.

From an abstract perspective, common to the IT systems mentioned above is the notion of concurrently running distributed components with local state that interact typically in an asynchronous way via messages. Thus any modeling framework for such systems should provide built-in capabilities for both state transitions and buffered communication. Many classical system modeling techniques focus either on state transitions like e.g. the B [Abr96] and Z [Spi92] notations or on interaction like the process algebra CSP [Hoa80] and the Pi-calculus [MPW92]. There are also efforts to combine the best of both approaches, e.g. 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 non-trivial formalisms. Moreover, theorem proving support respecting the structure of the mixed-style specifications seems not to be available.

There are at least three formalisms – with confusingly similar, or even equal, names – that pursue the approach of extending non-interactive automata with explicit input/output: both [MIB98] and [Jür02, §3] build their definitions on Gurevich's ASMs [Gur97] and consequently call the resulting formalisms *Interacting Abstract State Machines*. Common to both approaches is the use of *unordered* input/output buffers, which also constitutes the main conceptual difference to our approach. AutoFocus automata [HSSS96], as well as *Interactive Algebraic State Machines* [Jür03], provide essentially unbuffered clock-synchronous communication. *Focus* [BS01] uses the very abstract and mathematically elegant – yet on the other hand more difficult – model of stream-processing functions to describe the behavior of reactive systems. For neither of these formalisms mechanical theorem proving support is available.

There is a simple well-developed formalism that has been designed for modeling state-oriented asynchronous distributed computation from the outset: *I/O Automata (IOAs)* [LT89]. IOAs come with a mature meta theory that offers compositional refinement. System properties, both safety and liveness ones, may be described using temporal logics and proved manually or with suitable tools. The IOA approach has been implemented by Müller [Mül98] using the theorem prover Isabelle [Pau94]. This implementation supports not only interactive verification but also model checking [Ham99].

Thus IOAs seem to be a good candidate for the desired framework, but from our perspective they suffer from one severe drawback: 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, any such 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. This typically makes both modeling and verification rather cumbersome. Our solution in order to avoid these disadvantages is to add extra structure, essentially by designating parts of the local state of an automaton as input/output buffers and introducing transitions with simultaneous input/output inspired by AutoFocus automata [HSSS96]. The notion of ISMs implements these ideas.

In contrast to the article [OL02] emphasizing the application of ISMs to securiy modeling, the present article focusses on the ISM semantics and on verification techniques, in particular for analyzing authentication protocols.

2 Interacting State Machines

In this section, which is the core of the current article, we introduce the notion of Interacting State Machines both informally and mathematically. Then we introduce their graphical representation within AutoFocus and their textual representation within Isabelle/HOL. Finally, we give a semantic translation of ISMs to IOAs.

2.1 Concepts

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* one receiver listens to possibly many senders. An ISM *system* is the interleaved parallel composition of any number of ISM components where the state of the whole system is essentially the Cartesian product of the states of its components.

The *state* of an ISM consists of its input buffers and a 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[1] local *initial state*.

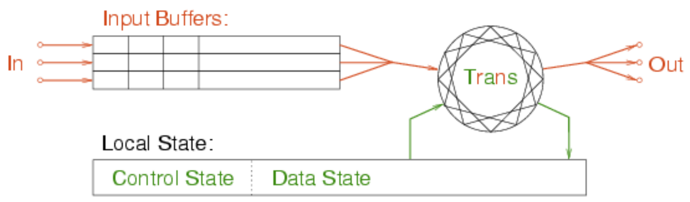

Fig. 1. ISM structure

Each ISM declares two sets of port names, one for input and the other for output. The input buffers are a family, indexed by the port names, of (unbounded) message FIFOs. Message exchange is triggered by any outputting ISM within the system or by the environment. 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 of an ISM may be processed by the ISM when it is ready to do so. This is done by user-defined *transitions*, which may be nondeterministic and can be specified in any relational style. Thus the user has the choice to define them 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 may include matching of messages in the input buffers – the ISM consumes as much input from its buffers as appropriate, makes a local state transition, and produces some output. The output is forwarded to the input buffers of all ISMs listening to the respective ports, which may result in direct or indirect feedback.

A *run* of an ISM or ISM system is any finite[2] (but unbounded) prefix of the sequence of configurations reachable from the initial configuration.

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

[2] 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 security properties are essentially safety properties: if at all they involve guarantees about the existence of future events, these typically involve timeouts.

Transitions of different ISMs that are composed in parallel are related only by the causality wrt. the messages interchanged. Execution gets stuck when there is no component that can perform any step. As typical for reactive systems, there is no built-in notion of final or "accepting" states.

2.2 Semantics

This subsection gives the logical meaning of ISMs in detail. This is meant as a precise reference and may be skipped for a first reading of the article.

Message Families Let M be the type of all messages potentially exchanged by ISMs and PN the type of port names. Then the *message families*, which are used to denote both input buffers and output patterns, have type $MSGs = PN \to M^*$ where M^* is any finite sequence of elements of M. The symbol $\not\Box$ denotes the empty message family $\lambda p.\ \langle\rangle$, the term $dom(m)$ abbreviates $\{p.\ m(p) \neq \langle\rangle\}$, i.e. the domain of a message family m, and the infix operation $m\ .@.\ n$ denotes pointwise concatenation $\lambda p.\ m(p)\ @\ n(p)$ of two message families m and n.

States and Transitions The type of an ISM state is $STATE(\Sigma) = MSGs \times \Sigma$ where the parameter Σ stands for the type of the local state. The set of transitions has type $TRANS(\Sigma) = \wp(STATE(\Sigma) \times MSGs \times STATE(\Sigma))$. Each of its elements has the form $((i,\sigma), o, (i',\sigma'))$ and means that the ISM can perform a step from local state σ to σ', taking the current input buffer contents i to i' (thus consuming as much input as required) and producing output o. Recall that i, i' and o each denote whole families of message FIFOs.

Single Automata An ISM is given as a quadruple $a = (In, Out, \sigma_0, Trans(a))$ of type $ISM(\Sigma) = \wp(PN) \times \wp(PN) \times \Sigma \times TRANS(\Sigma)$ where

- In is the set of input port names
- Out is the set of output port names
- σ_0 is the initial local state
- $Trans(a)$ is the transition relation

Such a definition is *well-formed* iff all the port names actually used in the transitions for input or output are contained in the sets In or Out, respectively. Note that In and Out may overlap, which enables direct feedback.

Runs The runs $Runs(a) \in \wp((MSGs \times STATE(\Sigma))^*)$ of an ISM a are finite sequences of configurations and is inductively defined as

$$\frac{dom(in) \subseteq In}{\langle(\not\Box, (in, \sigma_0))\rangle \in Runs(a)}$$

$$\frac{\begin{array}{c} cs^\frown(o,(b,\sigma)) \in Runs(a) \\ ((b,\sigma), o', (b',\sigma')) \in Trans(a) \\ dom(in) \subseteq In \end{array}}{cs^\frown(o,(b,\sigma))^\frown(o',(b'\ .@.\ in, \sigma')) \in Runs(a)}$$

The operator $^\frown$ appends elements to a sequence. ISM traces have the form $\langle(\not\!\!\!\square, (in_0, \sigma_0)), (o_1, (b_1 \ .@.\ in_1, \sigma_1)), (o_2, (b_2 \ .@.\ in_2, \sigma_2)), \ldots\rangle$ where each element of the sequence consists of the output, the input buffer contents, and the local state of the current step. Note that in each step the environment can provide arbitrary input *in* for the next step of the ISM. All ISM output is visible in the trace, whereas output not used for internal communication or feedback (within parallel composition, as described below) is discarded.

Parallel Composition The *parallel composition* $\|_{i \in I} a_i$ (with global input buffers) of a family of ISMs $A = (a_i)_{i \in I}$ is an ISM defined as the quadruple $(\mathit{AllIn} \backslash \mathit{AllOut},\ \mathit{AllOut} \backslash \mathit{AllIn},\ S_0,\ \mathit{PTrans}(A))$:

- $\mathit{AllIn} = \bigcup_{i \in I} \mathit{In}_i$
- $\mathit{AllOut} = \bigcup_{i \in I} \mathit{Out}_i$
- $S_0 = \Pi_{i \in I} (\sigma_0)_i$ is the Cartesian product of all initial local states
- $\mathit{PTrans}(A) \in \mathit{TRANS}(\Pi_{i \in I} \Sigma_i)$ is the parallel composition of the transition relations, defined as

$$\frac{\begin{array}{c} j \in I \\ ((b, \sigma), o', (b', \sigma')) \in \mathit{Trans}(a_j) \end{array}}{((b, S[j \mapsto \sigma]), o'_{|PN \backslash \mathit{AllIn}}, (b' \ .@.\ o'_{|\mathit{AllIn}}, S[j \mapsto \sigma'])) \in \mathit{PTrans}(A)}$$

$S[j \mapsto \sigma]$ denotes the replacement of the j-th component of the tuple S by σ. $m_{|C}$ denotes the restriction $\lambda p.\ \mathit{if}\ p \in C\ \mathit{then}\ m(p)\ \mathit{else}\ \langle\rangle$ of the message family m to the set C. The subterm $o'_{|PN \backslash \mathit{AllIn}}$ denotes those parts of the output o' provided to the environment, while $o'_{|\mathit{AllIn}}$ denotes the internal output to peer ISMs or feedback, which is added to the current buffer contents b'.

A parallel composition is well-formed iff all its components are well-formed. Note that there are no inter-component restrictions. This means in particular that inputs of different components may overlap (which leads to competition on inputs without fairness guarantees) and outputs may overlap as well (which leads to nondeterministic interleaving of outputs). An ISM system is called *closed* if $\mathit{AllIn} = \mathit{AllOut}$, i.e. there is no interaction with the environment.

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 the runs of a parallel composition for simplicity also directly (yet non-compositionally) as $\mathit{CRuns}(A) \in \wp((\mathit{STATE}(\Pi_{i \in I} \Sigma_i))^*)$ where we do not include ISM output in the trace:

$$\frac{\mathit{dom}(in) \subseteq \mathit{AllIn} \backslash \mathit{AllOut}}{\langle(in, S_0)\rangle \in \mathit{CRuns}(A)}$$

$$\frac{\begin{array}{c} j \in I \\ cs^\frown(b, S[j \mapsto \sigma]) \in \mathit{CRuns}(A) \\ ((b, \sigma), o', (b', \sigma')) \in \mathit{Trans}(a_j) \\ \mathit{dom}(in) \subseteq \mathit{AllIn} \backslash \mathit{AllOut} \end{array}}{cs^\frown(b, S[j \mapsto \sigma])^\frown(b' \ .@.\ o'_{|\mathit{AllIn}} \ .@.\ in, S[j \mapsto \sigma']) \in \mathit{CRuns}(A)}$$

One can show easily that running ISMs directly in parallel is equivalent to first combining the components in parallel, then running the system and projecting away the output from its trace: $\mathit{CRuns}(A) = \{\mathit{map}_{\pi_2}(cs) \mid cs \in \mathit{Runs}(\|_{i \in I} a_i)\}$.

2.3 Graphical Representation

When designing and presenting system models, a graphical representation is very helpful since it gives a good overview of the system structure and a quick intuition about its behavior. This is particularly important in an industrial setting: models are developed in collaboration with clients and documented for their further use, where strong familiarity with formal notations cannot be assumed. Therefore we have designed the structure of ISMs in a way that they can be easily displayed using an already available graphical tool, in this case AutoFocus.

AutoFocus [HSSS96] is a freely available prototype CASE tool for specification and simulation of 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 is visualized using *extended event traces (EETs)*.

As an illustrating example, take two figures from our model of Lowe's fix of the classical Needham-Schroeder public-key authentication protocol [Low96]. This model, which we call *NSL*, will be described in more detail in §3.

The system structure diagram in Figure 2 shows the four components with their local variables and the named connections between them, all including type information. The meaning of the diagram, i.e. the mapping to the ISM semantics, should be obvious.

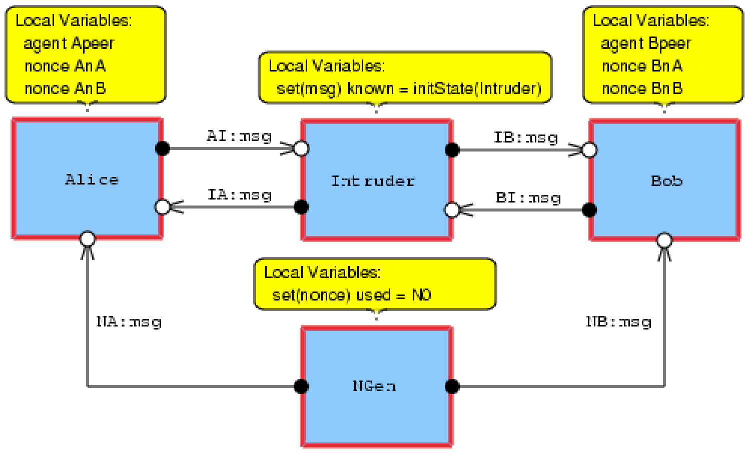

Fig. 2. NSL System Structure Diagram

Concerning the syntactic structure of systems, AutoFocus automata are richer than ISMs, which we have kept as basic as possible in order to simplify their semantics and to alleviate verification:

– We merge the AutoFocus notions of channels and ports into the notion of ports. The motivation for the more complex AutoFocus notions is easy re-use of components in different contexts. One can achieve an analogous effect by renaming of ports where required.

– AutoFocus automata may be hierarchical, which can be simulated by hierarchical use of the parallel composition operator and renaming in order to hide internal connections.

The state transition diagram in Figure 3 shows the three control states of the ISM `Alice` and the transitions between them, which have the general format precondition : inputs : outputs : assignments. Each input is given by a port name, the ? symbol, and a message pattern, while each output is given by a port name, the ! symbol, and a message value. A black bullet marks the initial state.

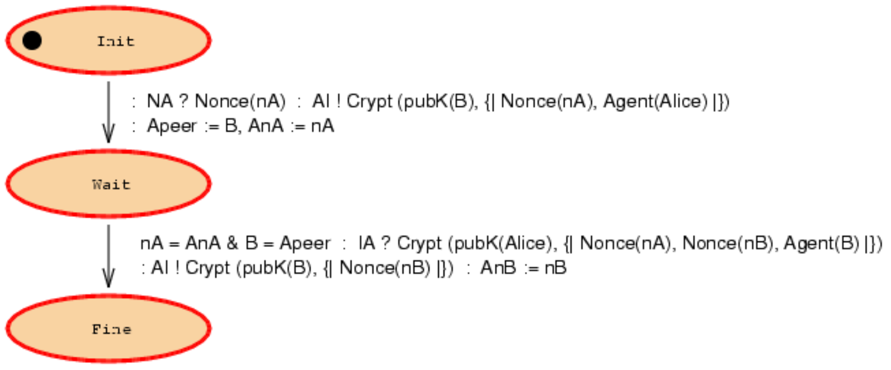
Init
: NA ? Nonce(nA) : AI ! Crypt (pubK(B), {| Nonce(nA), Agent(Alice) |})
: Apeer := B, AnA := nA
Wait
nA = AnA & B = Apeer : IA ? Crypt (pubK(Alice), {| Nonce(nA), Nonce(nB), Agent(B) |})
: AI ! Crypt (pubK(B), {| Nonce(nB) |}) : AnB := nB
Fine

Fig. 3. NSL State Transition Diagram: `Alice`

Concerning state transitions, the expressiveness of ISMs is higher than the one of AutoFocus automata because transition rules may be generic in the sense that each of them can describe a whole family of transitions:

– multiple source and/or destination control states
– non-constant port names, offering limited support for dynamic topologies
– unbounded nondeterminism concerning the values used for outputs and/or assignments
– changes to the local state can be given as arbitrary relational expressions

We use AutoFocus as a graphical front-end to our Isabelle implementation of ISMs. In a typical application of our framework, a user first "paints" ISMs using AutoFocus, saves them in the so-called *Quest* file format, and translates them into suitable Isabelle theory files, described in the next subsection, utilizing a tool program [Nan02,ON02].

Unfortunately, we cannot make use of the simulation, code generation and model checking capabilities of current AutoFocus and its back ends, which may be acquired by purchase from Validas [S^+]. This is because its underlying semantics is still clock-synchronous, due to the original emphasis of AutoFocus on embedded systems. In contrast, for the most of our applications, an asynchronous buffered semantics is more adequate. An alternative asynchronous semantics is currently under consideration also for future versions of AutoFocus. Anyway, the current semantic incompatibility is not a real obstacle to us since we are interested mainly in the graphical capabilities of AutoFocus and the AutoFocus syntax is general enough to cover also our deviating semantics.

2.4 Isabelle/HOL Representation

When aiming at system verification, one has the fundamental choice of using (automatic) model checking or (interactive) theorem proving techniques. We opt for the latter because the systems that we model are typically too complex for the capabilities of model checkers, and data abstraction techniques are either not applicable or would lead to counterintuitive modifications of the models. We employ Isabelle/HOL because of excellent experience with this tool.

Isabelle [Pau94] is a generic interactive theorem prover that has been instantiated to many logics, in particular the very practical *Higher-Order Logic (HOL)*. The only drawback of using Isabelle/HOL for applications like ours is the lack of dependent types: for each system modeled there is a single type of message contents into which all message data has to be injected, and the same holds for the local states of automata. Despite of this nuisance, we consider Isabelle/HOL the most flexible and mature verification environment available. Using it, security properties can be expressed easily and adequately and can be verified using powerful proof methods. Furthermore, Isabelle offers good facilities for textual presentation and documentation.

In order to represent ISMs in Isabelle theories in an adequately abstract way that has (almost) a one-to-one correspondence to the AutoFocus representation, we have designed a new theory section for Isabelle/HOL. This section is introduced by the keyword **ism** and has the following general structure[3]:

```
ism name =
  ports pn_type
    inputs   I_pns
    outputs  O_pns
  messages msg_type [buffers buffer_name]
  states    [glob_state_type]
   [control cs_expr0 :: cs_type]
   [data      ds_expr0[, ds_name] :: ds_type]
 [transitions
   (tr_name: [cs_expr -> cs_expr']
   [pre   (bool_expr)+]
   [in     (I_pn  I_msgs)+]
   [out   (O_pn O_msgs)+]
   [post ((lvar_name := expr)+ | ds_expr')]
 )+]
```

The meaning of the individual parts is as follows.

- *pn_type* is the Isabelle/HOL type of the port names, while *I_pns* and *O_pns* denote the set of input and output port names, respectively.
- *msg_type* is the type of the messages, which is typically an algebraic datatype with a constructor for each kind of message. The optional name *buffer_name*, which defaults to `ism_buffers`, is the name of a logical variable that may be used to refer to the contents of the input buffers within transition rules.

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

– The optional *glob_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, *glob_state_type* should be a free algebraic datatype with a constructor for each state type of the ISMs involved.
  cs_expr0 and *ds_expr0* specify the initial values of the control and data state, respectively, while *cs_type* and *ds_type* give their types. 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 `st`, may be used to refer to the whole data state within transition rules.

Transitions are given via named rules. The control state (if any) before and after the transition is specified by the expressions[4] *cs_expr* and *cs_expr'*.

Any expression within a rule may refer to the two logical variables mentioned above. In particular, the value of any local variable `lvar` of the ISM may be referred to by `st lvar` if `st` is the name of the data state variable. 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 transition.
– The **in** part gives a set of input port names *I_pn*, each in conjunction with a list *I_msgs* of message patterns expected to be present in the corresponding input buffer. When performing a transition, free variables in the patterns are bound to the actual values that have been input. Any input port not explicitly mentioned is left untouched.
– The **out** part gives a set of output port names *O_pns*, each in conjunction with an expression *O_msgs* denoting a list of values designated for output to the corresponding port. 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 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'* and suitable constraints in the preconditions.

An **ism** theory section as described above is translated to standard Isabelle concepts in a straightforward way using an extension to Isabelle/HOL, as described in [Nan02]. In particular, each ISM section is translated to a record 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 §2.2, in particular well-formedness, renaming, parallel composition, runs, and composite runs. Further auxiliary concepts are introduced as well, in particular reachability and several induction schemes related

[4] 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.

to ISM runs. Two of them will be given in §3.3 and §3.4. 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].

2.5 IOA Semantics

Next to the standard semantics of ISM runs given in §2.2, our Isabelle/HOL formalization [ON02] provides also two alternatives:

- the clock-synchronous semantics of AutoFocus mentioned in §2.3, which we do not further describe in this article, and
- a translation of ISMs to special instances of Input-/Output Automata [LT89], which yields an (essentially) equivalent semantics.

In this subsection we give a semi-formal description of the latter translation in order to show both the similarities and the differences of the two automata concepts.

The intuition behind the translation is that there is a one-to-one correspondence between IOAs and ISMs if each IOA is augmented by a pair of input and output buffers holding the built-in input buffers of the corresponding ISM and the ISM output not yet transmitted, respectively. Each ISM transition involving input of n messages and output of m messages is split into $n + 1 + m$ IOA actions (which may be interleaved with other actions): after the n input messages have arrived and have been stored via n (singleton) input actions of the IOA, an internal action performs the transition of the local state, consuming the n messages from the input buffers and appending the m output messages to the output buffers. These messages are later transmitted to their recipients via m (singleton) output actions. The resulting internal actions within any one IOA need not to be distinguished, while the internal actions of different IOAs must be kept distinct, which we achieve easily by augmenting them with the name of the ISM they are derived from. Recalling that IOAs communicate by synchronizing the sender and potentially many receivers on equal external actions, it becomes clear that each message sent or received by an ISM on a given port has to be represented on the IOA level by an external action holding both the port name and the message content.

More formally, a well-formed ISM C whose input and output ports do not overlap is translated to an IOA A as follows.

$sig(A)$, the action signature, is the triple $S = (in(S), out(S), int(S))$ where

$in(S)$, the set of input actions, contains all entities of the form $Extern\ pn\ v$, where pn is an input port name and v any message potentially transferred on the port pn.

$out(S)$, the set of output actions, contains all entities of the form $Extern\ pn\ v$, where pn is an output port name and v any message potentially transferred on the port pn.

$int(S)$, the set of internal actions, contains the single entity $Step\ C$, where C is the name of the ISM.

$states(A)$, the set of IOA states, contains all possible configurations, i.e. tuples of the form $(o, (i, \sigma))$ giving the current output and input buffer state as well as the local state of the ISM.

start(A)**,** the set of initial states, contains the single element $(¤, (¤, \sigma_0))$ representing the empty output buffers, empty input buffers, and the initial local state σ_0 of C.

steps(A)**,** the transition relation, consists of the following three sorts of transitions:

- for each input port name *pn* of C and any value v that is potentially input on the given port there is a transition labeled *Extern pn v* that appends v to the input buffer associated with *pn*.
- for each output port name *pn* of C and any value v currently at the head position of the output buffer associated with the port *pn*, there is a transition labeled by the output action *Extern pn v* that removes v from the buffer .
- for each (user-defined) transition of C there is a transition labeled *Step C* that reflects the change to the local state, removes from the input buffers all that is consumed, and appends all output to the output buffers.

part(A)**,** the equivalence relation used for describing fairness, is empty since we do not consider fairness, i.e. we define a so-called *safe IOA* [Mül98, Definition 2.2.2].

Note that the resulting automaton A is input-enabled, i.e. accepts input in any state, and has a well-formed signature S, i.e. $in(S)$, $out(S)$ and $int(S)$ are pairwise disjoint.

We have designed the format of the actions of A carefully such that it is possible to perform also the inverse translation from A to C, which essentially means projecting the input and output actions to input and output port names, removing the output buffers from the state, and keeping only the user-defined transitions where the ISM output is determined by the current difference of the output buffer states. It can easily be shown that the translation from ISMs to IOAs and back to ISMs is the identity mapping (even for ISMs that are not well-formed).

When considering finite executions only and disallowing the overlapping of input ports of ISMs composed in parallel (which would lead to multicasts according to the IOA semantics and non-determinism according to the ISM semantics), the direct ISM semantics and the one by translation to IOAs are equivalent. In particular, for any set of well-formed ISMs whose whose output ports do not overlap (which is a standard precondition for IOA composition), the given translation commutes with parallel composition on the ISM and IOA levels.

By the translation of ISMs to IOAs we get access to all the concepts available for IOAs. Reasoning about the properties of ISMs can be done on the level of IOAs, which means that we do not have to develop all proof methodology and tools from scratch. Thus in particular compositional reasoning on automaton refinement carries over to ISMs, as well as model checking support [Ham99]. Of course, it is desirable to transfer these concepts from the IOA level to the ISM level, which we do as far as our applications require and our development capacity allows.

We can conclude from this subsection that the expressive power of ISMs and IOAs is the same, while ISMs offer higher-level transitions and thus more structure that allows to model and verify reactive systems more adequately.

3 Authentication Protocol Verification

In this section we give an example of using the ISM framework for the verification of security properties in Isabelle/HOL. In order to allow for an easy comparison with other theorem proving approaches, in particular [Sch97], [Pau98] and [Coh00], we take the well-known example of the classical Needham-Schroeder public-key authentication protocol in the version fixed by Lowe [Low96].

3.1 Modeling

The honest agent *Alice* tries to initiate a single connection with *Bob* with the help of a nonce server *NGen* but in the presence of an attacker (according to the Dolev-Yao model) called *Intruder*. A diagram of the global structure of the system model as well as the state transition diagram for *Alice* is given in §2.3, while the textual representation of the four ISMs each describing the behavior of one system component can be found in the appendix. Our Isabelle/HOL formalization inherits the representation of messages and the intruder's capabilities from Paulson's protocol verfication theories explained in [Pau98]. In comparison to the versions given in [OL02], the model is slightly extended: we have added the local variable *AnB* to the state of *Alice* representing her view on the nonce received from her peer (which hopefully is the responder *Bob*).

In the remainder of this subsection we extend the comparison, given in [OL02], of our NSL model with Paulson's model [Pau98] to Schneider's model [Sch97]. There are major differences in two aspects. The first of them is obvious: Schneider's approach uses the CSP notation which does not have a notion of states, therefore system properties can be stated only with respect to message exchange. Sometimes even artificial messages have to be introduced just for specification and verification purposes. The same holds for Paulson's approach which is not based on CSP but also operates on message traces only. The second is more subtle: Schneider makes the implicit assumption that the nonces used by the honest agents are known in advance and that all of them are distinct. Paulson's model generates fresh nonces by choosing a value not already contained in the current trace of the system. Our model makes the computation and distinctness of nonces explicit by introducing the nonce server *NGen*.

3.2 Authentication Theorem

We aim to prove the most critical property of NSL, namely authentication of `Alice` to `Bob`, including session agreement. In our state-oriented ISM approach, this can be expressed very naturally, namely entirely on the basis of the states (in particular, knowledge and expectations) of the two honest agents:

```
theorem Bob_trusts_NSL: "Alice ∉ bad ⟶ (b,s)#cs ∈ CRuns(NSL) ⟶
(∃nA. Bob_state   s  = (Conn, (|Bpeer = Alice, BnA = nA, BnB = nB|))) ⟶
(∃(b',s')∈set cs.
(∃nA. Alice_state s' = (Fine, (|Apeer = Bob  , AnA = nA, AnB = nB|))))"
```

If *Alice* does not give away her private key and in some state of a run[5] of NSL *Bob* believes to be connected with *Alice* in a session identified by the nonce *nB*

[5] For technical reasons, ISM traces are constructed from right to left in Isabelle/HOL.

that he had brought up, then some time earlier Alice has indeed accepted a connection with Bob identified by the same nonce nB.

3.3 Invariant Proof

We prove the theorem using a variant of Schneider's rank function approach [Sch97]. Essentially, we show that as long as Alice does not reach the state of being happily connected to Bob in a session identified by nB (with we formulate as Alice_Fine_with Bob nB below), Bob cannot receive the final acknowledgment concerning nB (which is implied by the predicate noleak_all nB below) when trying to accept a connection with Alice (expressed by Bob_Resp_only_to Alice). Freshness of nonces, which we deal with in the next subsection, also plays a role, captured by the predicate Alice_Wait_nA nB. The key lemma just paraphrased can be stated formally in Isabelle/HOL as

```
lemma Bob_trusts_NSL_lemma: "Alice ∉ bad ⟶ cs ∈ CRuns(NSL) ⟶
     (∀(b,s)∈set cs. constrain nB s) ⟶ (∀c∈set cs. noleak_all nB c)"
```

where the predicate on the left-hand side of the implication is defined as

```
"constrain nB s ≡ ¬Alice_Wait_nA nB s ∧ Bob_Resp_only_to Alice s ∧
                  ¬Alice_Fine_with Bob nB s"
"Alice_Wait_nA nB s ≡ ∃as. Alice_state s = (Wait,as) ∧ AnA as = nB"
"Bobs_Resp_only_to A s ≡ ∀bs. Bob_state s = (Resp,bs) ⟶ Bpeer bs = A"
"Alice_Fine_with B nB s ≡ ∃nA. Alice_state s =
                              (Fine, (|Apeer = B, AnA = nA, AnB = nB|))"
```

and the right-hand side is a simplified version of the invariant given in [Sch97], adapted to the use within our stateful approach. It lifts the predicate noleak over all messages in the channels between the two agents and the intruder and over all information in the intruder's state:

```
"noleak_all nB (b,s) ≡ ∀p ∉ {NA, NB}. (∀m ∈ set (b p). noleak nB m)
                     ∧ (∀m ∈ Intruder_known s. noleak nB m)"
"noleak nB (Agent   a) = True"
"noleak nB (Nonce   n) = (n ≠ nB)"
"noleak nB (Key     k) = (Key k ∈ initState Intruder)"
"noleak nB {|m, n|}    = (noleak nB m ∧ noleak nB n)"
"noleak nB (Crypt k m) = (k=pubK Alice ∧ (∃n. m={|n,Nonce nB,Agent Bob|})
                          ∨ noleak nB m)"
```

The protocol-specific predicate noleak nB describes a superset of the messages actually transmitted by any agent (including the intruder) during runs of the constrained protocol, such that these messages cannot lead to sending Crypt (pubK Bob) (Nonce nB). This means in particular that nB must not be leaked to the intruder. In this case, one can prove that the intruder is unable to derive nB, i.e. noleak nB is an invariant of the intruder's behavior:

```
lemma init_noleak: "∀m ∈ initState Intruder. noleak nB m"
lemma synth_analz_maintains_noleak: "Alice ∉ bad ⟶
(∀m ∈ known. noleak nB m) ⟶ (∀m ∈ synth (analz known). noleak nB m)"
```

The great advantage of the given formulation of the key lemma is that it has the form of an invariant that can be proved by induction on the protocol runs.

Since both sides of the rightmost implication in the key lemma are universal quantifications on the configurations contained in a trace, the induction can be done rather conveniently. In this case we can make use of the following (derived) induction rule for a well-formed closed system of ISMs $A = (a_i)_{i \in I}$:

$$\frac{\begin{array}{c} P((\boxtimes, S_0)) \longrightarrow Q((\boxtimes, S_0)) \\ \forall j\ b\ \sigma\ o'\ b'\ \sigma'\ cs\ S.\ (j \in I\ \wedge\ ((b,\sigma), o', (b',\sigma')) \in \mathit{Trans}(a_j)\ \wedge \\ cs^\frown(b, S[j \mapsto \sigma]) \in \mathit{CRuns}(A)\ \wedge\ (\forall c \in (cs^\frown(b, S[j \mapsto \sigma])).\ P(c) \wedge Q(c)) \longrightarrow \\ P((b'\ .@.\ o', S[j \mapsto \sigma'])) \longrightarrow Q((b'\ .@.\ o', S[j \mapsto \sigma']))) \end{array}}{\forall cs \in \mathit{CRuns}(A).\ (\forall c \in cs.\ P(c)) \longrightarrow (\forall c \in cs.\ Q(c))}$$

Just observe that in order to apply this rule, one has to show only $Q((\boxtimes, S_0))$ and $Q((b'\ .@.\ o', S[j \mapsto \sigma']))$ while all other formulas are premises that may be taken advantage of. Proving the key lemma – including subproofs – takes about 70 lines of proof script (in the conventional semi-automatic tactics style of Isabelle) while applying it to get the main result takes about 20 lines.

3.4 Freshness Proof

We take about 40 extra lines of proof script to prove the freshness of nonces produced by *NGen*. The lemma interfacing this fact to the above proofs reads as

```
lemma Alice_Wait_not_BnB:
 "(b,s)#cs ∈ CRuns(NSL) ⟶ Bob_state s = (Resp, bs) ⟶
  (∀(b',s')∈set ((b,s)#cs). ¬Alice_Wait_nA (BnB bs) s')"
```

which means that the nonce that *Alice* had brought up and which she expects to get back from her peer is not the one brought up by *Bob*. We transform this proof goal to one stating that the nonce *Alice* expects can be only a value received on her port *NA*:

```
lemma Alice_Wait_NA: "cs ∈ CRuns(NSL) ⟶
 (∀(b,s)∈set cs. Nonce nB ∉ set b NA)) ⟶
 (∀(b,s)∈set cs. ¬Alice_Wait_nA nB s)"
```

which can be proved easily making use of the above induction rule again. The transformation requires eight straightforward freshness lemmas like

```
lemma NB_disjoint_NA_past: "(b,s)#cs ∈ CRuns(NSL) ⟹
 Nonce n ∈ set (b NB) ⟶ (b',s')∈set cs ⟶ Nonce n ∉ set (b' NA)"
```

all of which are proved in a very schematic way using the following simple induction rule for well-formed closed systems of ISMs $A = (a_i)_{i \in I}$:

$$\frac{\begin{array}{c} P(\langle\rangle,\ \boxtimes,\ S_0) \\ \forall j\ b\ \sigma\ o'\ b'\ \sigma'\ cs\ S.\ (j \in I\ \wedge\ ((b,\sigma), o', (b',\sigma')) \in \mathit{Trans}(a_j)\ \wedge \\ cs^\frown(b, S[j \mapsto \sigma]) \in \mathit{CRuns}(A)\ \wedge\ P(cs,\ b,\ S[j \mapsto \sigma]) \longrightarrow \\ P(cs^\frown(b, S[j \mapsto \sigma]),\ b'\ .@.\ o',\ S[j \mapsto \sigma'])) \end{array}}{\forall cs^\frown(b,s) \in \mathit{CRuns}(A).\ P(cs, b, s)}$$

All details of the proofs may be found at [ON02].

4 Conclusion and Future Work

We have introduced Interactive State Machines, a formalism for modeling and verifying the correctness and security of reactive state transition systems.

ISMs can be seen as high-level Input/Output Automata with the same expressiveness but significantly improved structuring. The ISM semantics can be translated to the IOA semantics, inheriting all their semantic concepts (except for fairness and liveness) as well as proof support. Yet for practicality reasons we prefer to define, respectively implement, them directly on the ISM level. Future work includes carrying over the concepts of refinement, compositionality and temporal logics as well as related proof methods and tools. For applications with restricted complexity, model checking support might be useful.

ISMs can also be viewed as variants of AutoFocus automata with an asynchronous buffered communication. Users of the approach can specify and present ISMs graphically as AutoFocus diagrams and then translate them to Isabelle theories. Alternatively, they may define ISMs also directly within Isabelle/HOL. The Isabelle representation enforces and supports fully formal – and thus maximally reliable – system modeling and verification. For verification, the powerful semi-automatic proof tools of Isabelle/HOL are available.

By the an example of using ISMs for authentication protocol analysis, our stateful approach (in contrast to e.g. Schneider's approach using CSP [Sch97], Paulson's inductive method [Pau98] or Cohen's TAPS [Coh00], which deal with communication events only) turns out to make system modeling as well as the formulation of security properties rather intuitive. Also the proofs provide more insights (via the invariants and freshness properties required), while due to the extra amount of detail, our proofs for NSL take more effort than Schneider's and in particular Paulson's and Cohen's.

Acknowledgments. We thank Volkmar Lotz, Thomas Kuhn, Haykal Tej, Jan Jürjens, Guido Wimmel and several anonymous referees for fruitful discussions and their comments on earlier versions of this paper.

References

Abr96. Jean-Raymond Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 1996.

BS01. Manfred Broy and Ketil Stø len. *Specification and development of interactive systems.* Springer, 2001.

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.

Coh00. Ernie Cohen. Taps: A first-order verifier for cryptographic protocols. In *13th IEEE Computer Security Foundations Workshop — CSFW'00*, pages 144–158, Cambridge, UK, 3–5 July 2000. IEEE Computer Society Press.

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

Gur97. Y. Gurevich. Draft of the asm guide. Technical Report CSE-TR-336-97, EECS Dept., University of Michigan, 1997.

Ham99. Tobias Hamberger. Integrating theorem proving and model checking in Isabelle/IOA. Technical report, TU München, 1999. http://www4.in.tum.de/reports/Ham-MC-99.html.

Hoa80. C. A. R. Hoare. Communicating sequential processes. In R. M. McKeag and A. M. Macnaghten, editors, *On the construction of programs – an advanced course*, pages 229–254. Cambridge University Press, 1980.

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.

Jür02. Jan Jürjens. *Principles for Secure Systems Design*. PhD thesis, Oxford University Computing Laboratory, Trinity Term 2002.

Jür03. Jan Jürjens. Algebraic state machines: Concepts and applications to security. In *Andrei Ershov 5th International Conference "Perspectives of System Informatics" (PSI'03)*, LNCS. Springer-Verlag, 2003.

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

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.

MIB98. M. Maia, V. Iorio, and R. Bigonha. Interacting Abstract State Machines. In *Proc. of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.

MPW92. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, September 1992.

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://home.in.tum.de/nanz/csthesis/.

OL02. David von Oheimb and Volkmar Lotz. Formal Security Analysis with Interacting State Machines. In Dieter Gollmann, Günter Karjoth, and Michael Waidner, editors, *Proc. of the* 7$^{\text{th}}$ *European Symposium on Research in Computer Security (ESORICS)*, volume 2502, pages 212–228. Spinger, 2002. http://ddvo.net/papers/FSA_ISM.html. A more detailed journal version is submitted for publication.

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

Pau94. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994. For an up-to-date documentation, see http://isabelle.in.tum.de/.

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

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

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

Spi92. J. Mike Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

A ISM Definitions for NSL

A.1 Auxiliary Definitions

```
datatype A_control = Init | Wait | Fine

record A_data =
  Apeer :: "agent"
  AnA   :: "nonce"
  AnB   :: "nonce"
consts A0 :: A_data

types A_state = "A_control × A_data"

datatype B_control = Idle | Resp | Conn

record B_data =
  Bpeer :: "agent"
  BnA   :: "nonce"
  BnB   :: "nonce"
consts B0 :: B_data

types B_state = "B_control × B_data"

types I_state = "msg set"

types N_state = "nonce set"
consts N0 :: "nonce set"

datatype state = AS A_state | IS I_state | BS B_state | NS N_state

datatype channel = AI | IB | BI | IA | NA | NB
```

A.2 Alice

```
ism Alice =
  ports channel
    inputs "{NA,IA}"
    outputs "{AI}"
  messages msg
  states   state
    control "Init"  :: A_control
    data    "A0", s :: A_data
  transitions
  Req: Init → Wait
    in  NA "[Nonce nA]"
    out AI "[Crypt (pubK B) ⦃Nonce nA, Agent Alice⦄]"
    post Apeer := B, AnA := nA
  Ack: Wait → Fine
    pre "nA = AnA s", "B = Apeer s"
    in  IA "[Crypt (pubK Alice) ⦃Nonce nA, Nonce nB, Agent B⦄]"
    out AI "[Crypt (pubK B) (Nonce nB)]"
    post AnB := nB
```

A.3 Bob

```
ism Bob =
  ports channel
    inputs "{NB,IB}"
    outputs "{BI}"
  messages msg
  states   state
    control "Idle"  :: B_control
    data     "B0", s :: B_data
  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)]"
```

A.4 Intruder

```
ism Intruder =
  ports channel
    inputs  "{AI, BI}"
    outputs "{IA, IB}"
  messages msg
  states   state
    data "initState Intruder", known :: "msg set"
  transitions
  Learn:
    pre "ch ∈ {AI, BI}"
    in   ch "[m]"
    post "insert m known"
  Utter:
    pre "ch ∈ {IA, IB}", "m ∈ synth (analz known)"
    out  ch "[m]"
```

A.5 Nonce Generator

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