Generic Interacting State Machines and their instantiation with dynamic features

David von Oheimb and Volkmar Lotz

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

Abstract. Interacting State Machines (ISMs) are used to model reactive systems and to express and verify their properties. They can be seen both as automata exchanging messages simultaneously on multiple buffered ports and as communicating processes with explicit local state.
We introduce generic ISMs, extending the ISM formalism with global state. We give a typical instantiation, namely support for dynamically changing communication. Other instantiations, e.g. an implementation of boxed mobile ambients, can be used alternatively or in combination, which demonstrates the flexibility of the framework. As an application example we model a simple multi-threaded client/server system.
ISMs and all their derivations are formally defined within the theorem prover Isabelle/HOL. The development, textual documentation, and verification of their applications is supported by Isabelle as well, and graphical design and documentation is available via the CASE tool AutoFocus. The conventional state-based approach, its expressiveness and flexibility, and freely available multi-level tool support makes our framework well-suited for practical formal system analysis even in an industrial setting.

Keywords: modeling, verification, composition, semantics, dynamic communication, mobile ambients, Interacting State Machines, Isabelle/HOL, AutoFocus.

1 Introduction

State-based approaches, e.g. [LT89,HLN+90,Spi92,HSSS96,Gur97,EHS97] have turned out to be an adequate means to model and analyze properties of interest in many of today's IT systems, including communication networks, database systems, and industrial control systems. In particular, the authors have introduced basic Interacting State Machines (ISMs) [OL02,Ohe02] and successfully applied them to security analysis, ranging from the specification and validation of security requirements with respect to very abstract system models to verification of low-level protocols. ISMs can intuitively be seen as a variant of I/O automata [LT89] offering high-level transitions allowing for simultaneous, buffered I/O on multiple ports. The resulting concepts have proved to be adequate for supplying formal security models for real-world smart card processor systems and health-care applications. In particular, the recent version of the LKW security model for the Infineon SLE 66 smart card chip has been developed using ISMs and their Isabelle [Pau94] tool support, see [OL02] for details.

However, ISMs as introduced so far lack expressiveness wrt. system dynamics. These may occur in a variety of flavors, comprising varying communication interfaces, the activation and deactivation of processes, and changes to the visibility context or execution environment of a component. All of these aspects are of practical relevance, e.g. in a middle-ware system where objects get to know additional communication channels by requesting a directory object, a multi-threaded system where the system components, along with their ports, are created and terminated dynamically, or a mobile agent system where the current execution environment determines the communication abilities of a hosted agent.

Driven by the above motivation, we generalize ISMs, introducing global state and commands that generic ISMs can execute in order to change it. By instantiating generic ISMs in a suitable way, one can handle dynamic communication interfaces as well as dynamic component contexts. Such extensions may be hierarchical or orthogonal, or they may combine other extensions while interrelating features as appropriate. The global state may express e.g. port ownership and the activation status of ports and ISMs, with the commands allowing to change port ownership and to (de-)activate ISMs and ports. We thus arrive at dynamic ISMs (dISMs). Alternatively, we may borrow the concepts of boxed ambients [BCC01] to treat dynamic contexts. Doing so, the global state is given by an ambient structure whose nodes refer to those ISMs that share the same administrative domain and thus can interact with each other. Commands include introducing and deleting contexts and moving them around. The resulting automata are called Ambient ISMs (AmbISMs). By combining the two concepts in the appropriate way, we arrive at dynamic Ambient ISMs (dAmbISMs). Figure 1 shows how the just mentioned dynamic extensions relate. The advantage of this "construction kit" approach is flexibility: the user may select either one of the two styles of dynamics (if not both of them are required) or their combination.

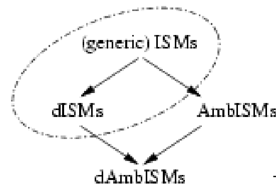

Fig. 1.

Generic ISMs (and their descendants) are supported by the same tools as basic ISMs: in a typical application of our framework, the user first specifies a system graphically with the CASE tool AutoFocus [HSSS96], then translates the model to theories of the theorem prover Isabelle/HOL [Pau94] using a tool program [Nan02,ON02], and then uses the facilities of Isabelle for conducting proofs and for textual documentation.

In this paper, we define generic and dynamic ISMs in detail and just introduce the concepts of ambient and dynamic ambient ISMs. The formal definition of the latter is subject of an accompanying paper [KO03]. The present paper is structured as follows. §2 formally defines generic ISMs and describes their representation with AutoFocus and Isabelle/HOL. In §3, we informally describe the different extensions and lay out the foundations of their semantics. A full definition of dynamic ISMs is given in §4. §5 contains a typical example of their use, namely a model of a multi-threaded client-server system where threads are activated dynamically on demand, and §6 comments on related and future work.

2 Generic Interacting State Machines (ISMs)

In §2.1 we introduce the notion of *generic ISMs*, which is a generalized (and partially simplified) version of the original ISM notion [Ohe02]. §2.2 gives the details of the semantics, which (as well as §4) may be safely skipped by readers solely interested in ISM application. Next, we describe how ISMs are represented as AutoFocus diagrams (in §2.3) and in Isabelle/HOL theories (in §2.4).

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*, (typically) one receiver listens to possibly many senders.

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*. It may hold additional *global state*, which may be affected by *commands* contained in the transitions of any (sub-)component, yet is not directly visible in the transitions.

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[1] local *initial state*.

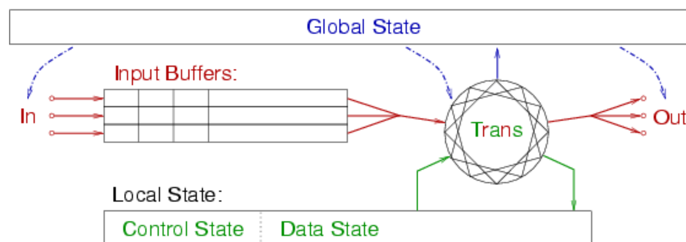

Fig. 2. ISM structure

The input buffers of an ISM are a family of (unbounded) message FIFOs, indexed by port names. The buffers are not actually part of an ISM but exist merely as intermediate data structures within parallel composition during ISM runs. Input buffers can (but in most applications should not) be shared among ISMs, which leads to competition on the input without fairness constraints.

[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.

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, issues a list of commands affecting the global state, 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, and by the effects of commands on the global state which may in effect block certain transitions. 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.

2.2 Semantics

This subsection gives the logical meaning of (generic) 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[2] 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 $\varnothing$ 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: $(m \ .@.\ n)(p) = m(p) \mathbin{@} n(p)$

States and Transitions Let $\mathcal{C}$ be the type of commands. Then the set of ISM transitions has type $TRANS(\mathcal{C}, \Sigma) = \wp((MSGs \times \Sigma) \times \mathcal{C} \times (MSGs \times \Sigma))$ 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 $((i, \sigma), c, (o, \sigma'))$ and means that the ISM can (nondeterministically) perform a step from local state σ to σ', consuming input i, executing command c, 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.

Elementary ISMs An ISM is given as a quadruple[3] $a = (In(a), Out(a), \sigma_0(a), Trans(a))$ of type $ISM(\mathcal{C}, \Sigma) = \wp(\mathcal{P}) \times \wp(\mathcal{P}) \times \Sigma \times TRANS(\mathcal{C}, \Sigma)$ 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

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

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

[2] Recall that output buffers are not required.

[3] 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, ...

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

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

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

The operator $^\frown$ appends elements to a sequence. Commands c are ignored here as we consider global state only for composite runs.

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.

The *parallel composition* $\|_{i \in I} A_i$ of a family of ISMs $A = (A_i)_{i \in I}$ is an ISM of type $ISM(\mathcal{C}, CONF(\Pi_{i\in I}\Sigma_i))$ 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 input buffers. The composite ISM is defined as the quadruple $(AllIn(A) \backslash AllOut(A),\ AllOut(A) \backslash AllIn(A),\ (\varnothing, S_0(A)),\ PTrans(A))$ where

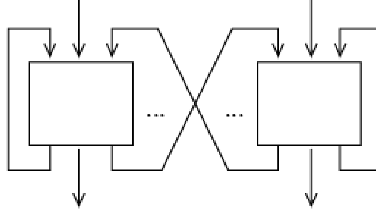

Fig. 3. General pattern of feedback within parallel composition

- $AllIn(A) = \bigcup_{i \in I} In(A_i)$
- $AllOut(A) = \bigcup_{i \in I} Out(A_i)$
- $\varnothing$ denotes the initially empty input 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 $TRANS(\mathcal{C}, CONF(\Pi_{i \in I} \Sigma_i))$ 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:

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

where

- $S[j:=\sigma]$ denotes the replacement of the j-th component of the tuple S by σ
- $m_{|P}$ denotes the restriction $\lambda p.\ \mathsf{if}\ p \in P\ \mathsf{then}\ m(p)\ \mathsf{else}\ \langle\rangle$ 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

Note that commands c are simply forwarded to the outer level of transitions.

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 below definition is generic wrt. ISM commands and the global state Γ, such that it may be used without further extension also for the specialized styles of ISMs defined in the following sections. Since the above definition of parallel

composition is generic wrt. ISM commands as well, it may be used in combination with composite runs to describe inner (possibly nested) levels of parallel composition.

For handling global state changes, composite runs have three parameters:

- a function $As(\gamma) = (As(\gamma)_i)_{i \in I(\gamma)}$ yielding an ISM family for any global state γ, which enables dynamic changes to the ISM system
- the initial global state γ_0
- a transition relation $gtrans(j)$ that takes as its parameter the index of the ISM whose transition is currently performed and yields a transition between the global pre-state γ, the command c, and the global post-state γ'

The set of all possible *composite runs* is denoted by $CRuns(As, \gamma_0, gtrans)$ and has type $\wp((CONF(\Gamma \times \Pi_{i \in I} \Sigma_i))^*)$ corresponding to the generic ISM type $ISM(\mathcal{C}, \Gamma \times \Pi_{i \in I} \Sigma_i)$. Its elements are finite sequences of configurations, inductively defined as

$$\frac{}{\langle(\not\square, (\gamma_0, S_0(As(\gamma_0))))\rangle \in CRuns(As, \gamma_0, gtrans)}$$

$$\frac{\begin{array}{c} j \in I(\gamma) \\ cs ^\frown (i \mathbin{.@.} b, (\gamma, S[j := \sigma])) \in CRuns(As, \gamma_0, gtrans) \\ ((i, \sigma), c, (o, \sigma')) \in Trans(As(\gamma)_j) \\ mdom(i) \subseteq In(As(\gamma)_j) \cap AllOut(As(\gamma)) \\ mdom(o) \subseteq Out(As(\gamma)_j) \cap AllIn(As(\gamma)) \\ (\gamma, c, \gamma') \in gtrans(j) \end{array}}{cs \frown (i \mathbin{.@.} b, (\gamma, S[j := \sigma])) \frown (b \mathbin{.@.} o, (\gamma', S[j := \sigma'])) \in CRuns(As, \gamma_0, gtrans)}$$

Note that the changes to the local state σ and the global state γ are independent of each other, except that the transition $Trans(As(\gamma)_j)$ may block $gtrans(j)$ and vice versa. The restrictions on the input and output domains are the dynamic counterparts for the static well-formedness of the family components and the closedness of the system. They ensure in particular that ISMs can use only ports they are allowed to according to their I/O interface which may depend on the current global state[4]. An ISM family function – together with the initial global state and the global transition associated with it – that fulfills the restrictions on the input and output domains already by its construction is called *dynamically closed*.

Traces of composite runs have the form $\langle(\not\square, (\gamma_0, S_0(As(\gamma_0)))), (b_1, (\gamma_1, S_1)), (b_2, (\gamma_2, S_2)), \ldots\rangle$ where each element of the sequence is a nested tuple of the current input buffer contents, the current global state, and the Cartesian product of all the currently relevant local states.

One can show that composite runs of any closed family $As(\gamma)$ of well-formed ISMs are equivalent to the runs of the parallel composition of the same family

[4] Note that this restriction is defined wrt. γ and not γ' for both input and output, which makes the definition technically slightly simpler. A viable alternative would be to use instead γ' for restricting the output, which would implement the idea that the effects of the command c are already visible to the output operations.

if the global transition relation is the identity and the global state is projected away from the traces:
$wf_isms(As(\gamma)) \;\wedge\; closed(As(\gamma)) \;\longrightarrow\; Runs(\|_{i \in I(\gamma)} As(\gamma)_i) =$
$\{map_{(\lambda(b,\gamma,\sigma).\ (b,\sigma))}(cs) \mid cs \in CRuns(As, \gamma, (\lambda i.\ \{(\gamma, c, \gamma') | \ \gamma = \gamma'\}))\}$

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.

Unfortunately, we do not have a graphical tool available that could cover the dynamic port connections and ambient structures described in the following sections. Nevertheless, we have designed the structure of generic ISMs in a way such that their basic features can be displayed using the CASE tool AutoFocus.

One may use AutoFocus as a graphical front-end to our Isabelle implementation of ISMs: the user first specifies ISMs using AutoFocus and translates them into suitable Isabelle theory files, described in §2.4 below, utilizing a tool program [Nan02,ON02]. ISM commands, which are not directly supported by AutoFocus automata, may be simulated by output to special channels.

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 a multi-threaded client/server architecture: a server spawns a new working thread for each request received from a client. The system structure diagram in Figure 4 shows one client, the server, and two threads 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.

The state transition diagram in Figure 5 shows the three control states of a `Thread` ISM 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. The initial control state is marked with a black bullet. The output to the special port `cmd` represents dynamic ISM commands as described in §4. The example will be described in detail in §5.

The simulation, code generation and model checking capabilities of AutoFocus cannot be used for (our extended versions of) ISMs because its underlying semantics is clock-synchronous and does not deal with commands and global state. Anyway, if one is interested mainly in the graphical capabilities of AutoFocus, the AutoFocus syntax is general enough to cover most aspects of ISMs.

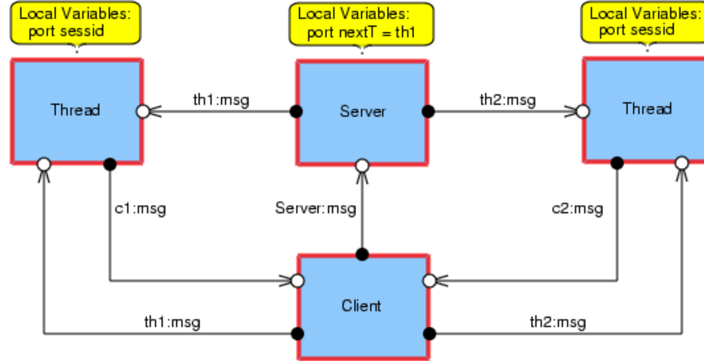

Fig. 4. Client/Server System Structure Diagram

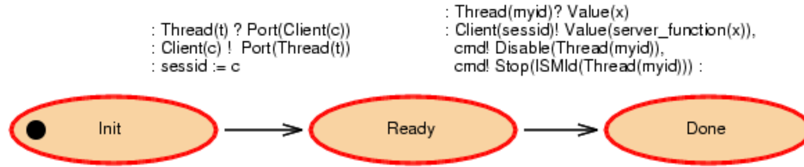

Fig. 5. Client/Server State Transition Diagram: `Thread`

2.4 Isabelle/HOL Representation

When aiming at rigorous formal modeling or even system verification, tools performing syntactic checks, type checks, and mechanized proofs are essential. We employ the theorem proving system 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)*. Despite of one nuisance[5], we consider Isabelle/HOL the most flexible and mature modeling and verification environment available. Using it, system 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.

ISMs can be defined in special Isabelle theory sections. Their standard interpretation is the meta theory described in §2.2. It is implemented by an Isabelle plug-in [Nan02] in connection with a library of Isabelle theories. The Isabelle/HOL representation of ISMs has essentially a one-to-one correspondence to the AutoFocus representation described above.

[5] The only drawback of 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 ISM states. The alternative prover PVS supports dependent types, but on the other hand it is less flexible, in particular, user-defined theory sections are not possible.

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
 [commands cmd_type [default cmd_expr']]
   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)+ ]
   [cmd    cmd_expr]
   [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 its corresponding type *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. If ports can be changed dynamically, like with dynamic ISMs, the sets given here specify the initial or maximal interface.
- 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 *cmd_type* specifies the type of ISM commands. It must be given if commands are used in the transitions. The optional default command *cmd_expr'*, which typically is the empty list of commands, can be used to shorten the specification of transitions that do not actually issue commands.
- 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.

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

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'*.

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 **cmd** part gives the ISM command *cmd_expr* associated with the current transition. Such a command can be given in each transition if the **commands** subsection is present.
- 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 §2.2, 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 §5.

[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.

3 Extensions

In this section we give a conceptual overview of the instantiations of generic ISMs available so far, namely by dynamic ports and running state of ISMs, by ambient structures, and the combination of these features.

3.1 Dynamic ISMs

Dynamic ISMs (dISMs) are an instantiation of generic ISMs offering dynamic creation, transfer, enabling and disabling of ports. They also offer activation and deactivation of dISMs. This may be used to emulate ISM creation and deletion, provided that all possible ISMs of the desired form are part of the system. Note that "genuine" creation and deletion would not only be beyond the limits of the underlying logic and its type system, but also less general: it would not give the possibility to "reawaken" ISMs. These dynamic features show the power of the generalization of ISMs. An application example that makes use of all these features is given in §5.

A system of dynamic ISMs uses the global state to keep track of the currently running dISMs, enabled ports, and port ownership. Changes to this state are made by members of the system issuing suitable commands: a dynamic ISM may request that a dISM not yet running is activated or a running dISM (including itself) is stopped. Moreover, a dynamic ISM may create a new port and become its initial owner. An owner of a port may receive input on the port, allow or forbid others to output to it, or convey it to any other dISM. The facility to enable or disable ports can be used to model e.g. flow control.

3.2 Ambient ISMs

An instantiation of generic ISMs quite different from dynamic ISMs are *Ambient ISMs (AmbISMs)* [KO03]. They give a novel form of operational semantics to the ambient calculus [CG98] where we extend the ability to communicate along the lines of boxed ambients [BCC01]. Most importantly, by combining ambient processes with ISMs, we introduce a concept of process state.

Ambients are nested administrative domains that contain *processes* which (in our case) are ISMs. As usual, the ambient structure determines the ability of the processes to communicate with each other. Ambients are *mobile* in the sense that an ISM may move the ambient it belongs to, together with all ISMs and sub-ambients contained in it, out of the parent ambient or into a sibling. Moreover, an ambient may be deleted ("opened") such that its contents are poured into the surrounding ambient, or a new ambient may be created as a child of the current one. Finally, (new) ISMs may be assigned to ambients.

All these operations are implemented by ISM commands manipulating a particular instantiation of the global state, which is given by a tree structure representing the ambient hierarchy.

In the ambient literature, ambient operations are called *capabilities* since their "possession" can be seen as a qualification to perform the respective action.

Semantically speaking, the qualification simply boils down to knowing the name of the ambient involved.

3.3 Dynamic Ambient ISMs

As the name suggests, *dynamic Ambient ISMs (dAmbISMs)* [KO03] combine dynamic ISMs and Ambient ISMs.

Dynamic Ambient ISMs inherit port handling and dAmbISM (de-)activation from dynamic ISMs and ambients from Ambient ISMs. The concepts are mostly orthogonal, except for one new feature: it is reasonable to offer the operations that affect other dAmbISMs j (by activating or deactivating them or conveying ports to them) only to dAmbISMs that are in the vicinity of j, by restricting the respective dISM commands. We call this *locality* of dAmbISM manipulation.

We have taken care in designing dynamic ISMs and Ambient ISMs such that their combination is painless both on implementation and application levels.

4 Semantics of Dynamic ISMs

In this section, we define the semantics of dynamic ISMs in detail and comment on some of their properties. In doing so, we build immediately on the definitions given in §2.2. Readers focusing on ISM application may just note the six dynamic ISM commands (with their obvious parameters) and skip the details.

Dynamic State and Commands The global state of dynamic ISMs has the form $\delta = (running(\delta), enabled(\delta), owned(\delta))$, instantiating the generic global state Γ to $dSTATE = \wp(\Im) \times \wp(\mathcal{P}) \times (\Im \to \wp(\mathcal{P}))$ where $\Im$ is the type of dISM identifiers. $running(\delta)$ is the set of dISMs currently active, $enabled(\delta)$ the set of ports currently enabled, and $owned(\delta, i)$ the set of ports currently owned by the dISM i.

The ISM type parameter $\mathcal{C}$ gets instantiated to dynamic ISM commands $dCMD^*$ where $dCMD = \{Run(i) | i \in \Im\} \cup \{Stop(i) | i \in \Im\} \cup \{New(p) | p \in \mathcal{P}\} \cup \{Convey(p, i) | p \in \mathcal{P} \wedge i \in \Im\} \cup \{Enable(p) | p \in \mathcal{P}\} \cup \{Disable(p) | p \in \mathcal{P}\}$.

Dynamic Transitions Let i be the current dISM and js be the set of dISMs that it is allowed to start or stop or convey ports to. The global transition relation $dTrans(js, i)$ is defined as $\{(\delta, dcmds, \delta') \mid i \in running(\delta) \wedge \delta \xrightarrow{i:js:dcmds}{}^{*} \delta'\}$ where the single-step command execution relation $\delta \xrightarrow{i:js:dcmd} \delta'$ means that the command $dcmd$ issued by i transfers the dynamic state δ to δ', as defined by the rules

$$\frac{j \notin running(\delta) \;\wedge\; j \in js}{\delta \xrightarrow{i:js:Run(j)} \delta(\!| running := running(\delta) \cup \{j\} |\!)}$$

$$\frac{j \in running(\delta) \;\wedge\; j \in js}{\delta \xrightarrow{i:js:Stop(j)} \delta(\!| running := running(\delta) \setminus \{j\} |\!)}$$

$$\frac{p \in owned(\delta, i) \ \wedge \ p \notin enabled(\delta)}{\delta \xrightarrow{i:js:Enable(p)} \delta (\!| enabled := enabled(\delta) \cup \{p\} |\!)}$$

$$\frac{p \in owned(\delta, i) \ \wedge \ p \in enabled(\delta)}{\delta \xrightarrow{i:js:Disable(p)} \delta (\!| enabled := enabled(\delta) \setminus \{p\} |\!)}$$

$$\frac{p \notin \bigcup i. \ owned(\delta, i)}{\delta \xrightarrow{i:js:New(p)} \delta (\!| owned := ((owned(\delta))[i := owned(\delta, i) \cup \{p\}] |\!)}$$

$$\frac{p \in owned(\delta, i) \ \wedge \ p \notin owned(\delta, j) \ \wedge \ j \in js}{\delta \xrightarrow{i:js:Convey(p,j)} \delta (\!| owned := ((owned(\delta))[j := owned(\delta, j) \cup \{p\}, \ i := owned(\delta, i) \setminus \{p\}]) |\!)}$$

where $_(\!|_ := _|\!)$ is the component update operator on tuples.

Marking non-existing dISMs as running is possible but harmless. A dISM j may be put into the running state only if it is not currently running and stopped only if it is currently running. Ports may be conveyed also to dISMs not currently running. Common to the $Run(j)$, $Stop(j)$, and $Convey(p, j)$ commands is that the range of ISMs j affected by them can be restricted by the set js. In the definition of *dCRuns* below, js is instantiated to the universal set (implying no restrictions), but in the definition of dynamic Ambient ISMs [KO03], js is used to implement a locality constraint.

A port p may be enabled only if it is not currently enabled and disabled only if it is currently enabled. Only a current owner may receive from, enable, disable, and convey a port. Freshness of a new port is guaranteed by requiring that it is not currently owned by any dISM. The definition of *set_In_Out* below implies that input may be received also from ports not currently enabled, while output may be sent only to enabled ports owned by currently running dISMs. The initial input interface of an ISM determines its initial port ownership, whereas the initial output interface serves as an upper limit of the output interface throughout the life of the ISM.

Composite Runs Finally, we instantiate the generic composite runs operator for ISMs according to the needs of dynamic ISMs: for any dISM family A and any set $r \subseteq I$ of ISM identifiers describing those dISMs that shall be running initially, $dCRuns(A, r)$ gives the (set of traces of) composite runs of dynamic ISMs. It has type $\wp((CONF(dSTATE \times \Pi_{i \in I} \Sigma_i))^*)$, corresponding to the dISM type $ISM(dCMD^*, dSTATE \times \Pi_{i \in I} \Sigma_i))$, and is defined as

$$dCRuns(A, r) \equiv CRuns(set_In_Out(A), init_dSTATE(A, r), dTrans(\overline{\emptyset}))$$

where

- $\overline{\emptyset}$ is the complement of the empty set, i.e. the universal set.
- $init_dSTATE(A, r) = (r, \overline{\emptyset}, (\lambda i. \ \mathsf{if}\ i \in I \ \mathsf{then}\ In(A_i) \ \mathsf{else}\ \emptyset))$ yields the initial dynamic state where the set of running dISMs is r, all ports (even those not yet existing) are enabled, and port ownership is according to the input interfaces of the members of A.

– $set_In_Out(A,\delta) = (A_i(\!|In := owned(\delta,i), Out := Out(A_i) \cap enabled(\delta) \cap \bigcup_{j \in running(\delta)} owned(\delta,j)|\!))_{i \in I}$ transforms the initial ISM family A according to the dynamic state δ by setting the input interface of each member i to the ports it currently owns and the output interface to the subset of the initial output ports that are currently enabled and owned by some running dISM j.

As a consequence of these definitions, and the generic definition of *CRuns*, a family A of dISMs runs as follows. Initially, a subset r of the members of A is active, all ports are enabled, and port ownership is determined by the corresponding input interfaces. According to this initial dynamic state δ_0 produced by *init_dSTATE*, a new dISM family A' is determined by *set_In_Out*. When a member of A' performs a transition, the dynamic commands contained in the transition transform the dynamic state to δ_1, from which the next dISM family A'' is determined and used for the next transition, and so on.

Basic Properties Since port ownership is used to defined the input interface of the ISMs, the notion of well-formedness of parallel compositions introduced in §2.2 means in the context of dynamic ISMs that port ownership is unique. This nice property is preserved by the dynamic commands, as can be seen easily: in the case of port creation, only one dISM becomes the owner of the new port, and in the case of port transfer, port ownership is removed from the initial owner and given to a single new owner. All other commands do not affect port ownership.

According to the definition of *CRuns*, the ports that a dynamic ISM is allowed to use in its transitions are determined by the initial dynamic state of the transition at hand. This implies for example that ports newly created by (the commands part) of a transition can be communicated to peer dISMs immediately (i.e. in the same transition), but cannot immediately be used for sending or receiving messages on it. Yet this is not a restriction of expressiveness because the port will be available for I/O in any further transitions, and communicating via a newly created port usually makes sense only if the port has already become known to some peer ISM, which typically incurs some delay anyway.

5 Application Example

We present a typical application of dynamic ISMs for modeling dynamically communicating systems: the multi-threaded client/server architecture introduced in §2.3. It demonstrates dISM activation and deactivation, port creation, port transfer, and disabling of ports.

Our translation tool converts the AutoFocus diagrams to an Isabelle theory as outlined in §2.4. Typically, the user then edits that theory file in order to enhance the presentation and augments it with commenting texts and proofs. We reproduce here the complete textual documentation of the resulting Isabelle theory, as automatically produced by the LaTeX documentation facility of Isabelle.

theory *ClientServer = ISM_package:* — including dISM definitions

The example consists of a client that concurrently opens two sessions with a server. The sessions are identified by the reply ports provided by the client. The server spawns a working thread for each connection request received, creates a port for the thread, conveys the port to it, sends the client port to the new thread port, and awaits any new requests. Each thread receives its own port as well as the client port it is responsible for and sends the thread port to the respective client. The client receives the thread port and uses it to send the value it wants to be processed. The thread receives the value, computes the response value, sends it to the client, disables its port, and stops itself. Finally, the client collects the responses from the two server threads.

First we define the type of ports and dISM identifiers and a mapping between the two. There is one server with one port, several threads with one port each, and one client with several ports. The tags (i.e., datatype constructors used in the definitions of *port* and *id*) help us to statically map ports to dISM identifiers, which we do using the auxiliary function *ISMId*. Furthermore, we define a type abbreviation *cs_cmds* for the instance of dynamic ISM commands used here.

```
typedecl sid  — session identifier
typedecl tid  — thread identifier
datatype port =  Server |  Thread tid |  Client sid
datatype id   = iServer | iThread tid | iClient
consts    ISMId :: "port ⇒ id"
primrec "ISMId  Server    = iServer"
        "ISMId (Thread t) = iThread t"
        "ISMId (Client c) = iClient"
types cs_cmds = "(id, port) dcmd list"
```

The type of user data is called *val*. The server threads perform a function (taking values to values) called *server_function* which is not further specified. Messages sent within the system consist of either a port name or a value.

```
typedecl val  — value
consts server_function :: "val ⇒ val"
datatype msg = Port port | Value val
```

The server has a data state holding the identity of the next thread to be created. A thread may be in one of three control states and has two local variables in its data state: the thread identifier and the client session identifier indicating the session the thread is engaged in. Clients have a control state but no data state. For technical reasons, namely the lack of dependent types in Isabelle/HOL, we have to construct the union type *state* of all different local ISM states which will be used in each **ism** section and the definition of the overall *System* below.

```
types Server_data = tid
datatype Thread_control = Init | Ready | Done
record Thread_data =
  sessid :: "sid"
```

```
datatype Client_control = Open | Send | Close | Halt
datatype state = SS Server_data
               | TS "Thread_control × Thread_data"
               | CS Client_control
```

The server dISM, as well as all other dISMs, declares the type *port* for its port interface, *msg* for the messages it sends and receives, and *cs_cmds* for the dISM commands it issues. It listens only to the port *Server* but potentially talks to all threads that may ever come into existence. The name of its data state is *nextT* identifying the next thread to be dispatched, with the initial value *th1*. We let the server pre-compute (and store for the next connection request) the identity of the next thread because this saves us from defining two transitions where in the first transition the server receives the client port, stores it and creates a new thread as well as a new port, and in the second transition the server sends the newly created port and the client port to the thread.

```
consts th1 :: tid
ism Server =
  ports port
    inputs  "{Server}"
    outputs "{Thread t |t. True}"
  messages msg
  commands cs_cmds
  states state
    data Server_data init "th1" name "nextT"  — next thread to be created
  transitions
  dispatch:
    pre "tp = Thread nextT"  — just used as an abbreviation mechanism
    in  "Server"         "[Port cl]"
    out "Thread nextT"  "[Port cl]"
    cmd "[Convey tp (ISMId tp), Run (ISMId tp), New (Thread th')]"
    — the thread identifier th' is fresh by the semantics of New
    post "th'"   — the new value of nextT is th'
```

The ISM definition of threads is parameterized with the thread identifier *myid*. The initial input interface is empty because the server supplies the thread port dynamically. The output interface is the set of (potentially) all clients ports. The thread holds its only local variable (with arbitrary initial value) in the data state variable *s* and sets it according to the port received in its first transition. The definition of this transition does not give a **cmd** subsection and thus makes implicit use of the default command, which is the empty list here.

```
ism Thread (myid::tid) =
  ports port
    inputs  "{}"  — port will be supplied by the Server
    outputs "{Client c |c. True}"
  messages msg
  commands cs_cmds default "[]"
  states state
    control Thread_control init "Init"
```

```
    data     Thread_data
  transitions
  "init":
    Init → Ready
    in  "Thread myid" "[Port (Client c)]"
    — the input message pattern Port (Client c) guarantees to the thread that
       the port received actually is a client port (where c identifies the session)
    out "Client c" "[Port (Thread myid)]"
    post sessid := "c"
  work:
    Ready → Done
    in  "Thread myid" "[Value x]"  — s denotes the current data state
    out "Client (sessid s)" "[Value (server_function x)]"
    cmd "[Disable (Thread myid), Stop (ISMId (Thread myid))]"
```

We declare two session identifiers used as the input interface of the client. The client may output to the server and any threads. It sends the two connection requests immediately one after the other to the server port, waits until it has received the thread ports on its two ports, uses the two thread ports to concurrently send two (arbitrary) request values, and collects the two responses. Note the use of the control state to serialize the I/O operations. This example client synchronizes its two sessions. Of course, one could add further clients whose (single or multiple) sessions do not interfere at all with the other sessions.

```
consts c1 :: sid
consts c2 :: sid
ism Client =
  ports "port"
    inputs  "{Client c1, Client c2}"
    outputs "{Server} ∪ {Thread c |c. True}"
  messages msg
  commands cs_cmds default "[]"
  states state
    control Client_control init "Open"
  transitions
 "open":
    Open → Send
    out Server "[Port (Client c1), Port (Client c2)]"
  send:
    Send → Close
    in "Client c1" "[Port (Thread t1)]", "Client c2" "[Port (Thread t1)]"
    out"Thread t1" "[Value x1]        ", "Thread t2" "[Value x2]"
  close:
    Close → Halt
    in "Client c1" "[Value y1]        , "Client c2" "[Value y2]"
```

The overall system maps the dISM identifiers to the corresponding dISMs. Note that the parallel composition already includes all the threads that may become active at some time. When defining the composite runs of the system, we specify that initially the client and the server, but no thread, is running.

constdefs

```
 System :: "(id, (cs_cmds, port, msg, state) ism) family"
"System  ≡ (λi. case i of iServer     ⇒ Server.ism
                        | iThread tid ⇒ Thread.ism tid
                        | iClient     ⇒ Client.ism,
            {iServer, iClient} ∪ {iThread t |t. True})"
 Runs :: "((port, msg, (id,port) dstate × (id ⇒ state)) conf list) set"
"Runs ≡ d_comp_runs System {iClient, iServer}"
```

The parallel composition of all dISMs in the system is (at least initially) well-formed, i.e. their inputs do not overlap:

theorem `wf_comp_System: "wf_comp System"`

The proof of this property is routine and essentially automatic.

The system components are (statically) well-formed if the input interfaces of the threads are not taken into account. The system is dynamically closed because the server augments the input interface of each thread with the `Convey` command *before* the thread is activated, the thread receives input only from the port conveyed to it, and all input and output operations of all system components have their counterparts within the system.

end

This ends our small application example. It should demonstrate that dynamic ISMs are adequate means to describe dynamically changing communication patterns and that the abstraction level of ISMs is high enough for focusing on the essential aspects of reactive systems and low enough for making the transition to the technical implementation straightforward.

6 Discussion

Since dynamic ISMs provide a stateful notion of both dynamic and reactive systems, it is interesting to compare them both with well-known notions of dynamic systems and with state automata used for modeling reactive systems.

The π-calculus [MPW92] and its descendants have a built-in notion of communication channels and handle dynamics by passing channel identifiers and instantiating channel variables. Restricting the calculus to a small number of basic concepts leads to a concise and very abstract formalism still bearing a rich meta-theory. This makes the π-calculus particularly suited to study general concepts, but lacks feature for adequately specifying complex industrial-scale systems. In particular, state information is cumbersome to encode, as is local computation: even basic data types like numbers and the operations on them, which naturally occur *within* processes, have to be translated to auxiliary processes that need to interact via extra channels — an utterly inadequate representation that renders practical applications incomprehensible. Moreover, communication is synchronous via global channels without support for simultaneous multiple I/O and ownership restrictions like in dISMs. There is some basic tool support for

verification but not for graphical design and documentation of complex specifications, as offered by AutoFocus. We conclude that the quite abstract π-calculus is well-suited for academic research, while the more operational-style dynamic ISM approach with its rather concrete structure tailored for stateful reactive systems makes practical system analysis easier to conduct and to understand.

In the area of state-based automata, there are approaches particularly addressing the complexity of industrial-scale systems like StateCharts, which are supported by the CASE tool StateMate [HLN+90]. Statecharts offer rich structuring notions like hierarchical states as well as a notion of time. On the other hand, the communication facilities offered by are rather basic: messaging is achieved via synchronous events that are distributed globally, i.e. in an undirected (and uncontrollable) way. Though the advanced concepts of Statecharts like hierarchical states could at least partially be encoded in the (dynamic) ISM setting, Statecharts seem to be more adequate in the area of modeling non-distributed systems. On the other hand, they do not offer asynchronous communication with controlled dynamics that is desirable to model systems like the one given in §5. Moreover, tool support for verification is very poor, which is partially due to the fact that the precise semantics of Statecharts (intentionally) had been left underspecified such that various interpretations exist.

Compared to dISMs, also most other state based automata approaches are more basic with respect to communication. Recall that the development of the ISM notion has been originally motivated by the need to extend I/O automata [LT89] with more advanced communication concepts.

There is other related work addressing the extension of state based approaches with particular aspects of dynamic behavior, e.g., [GR95,HS97,Zap02], however, we are not aware of any flexible approach, supported by CASE and verification tools, that can handle several kinds of dynamics in combination.

The ISM instantiations with dynamic features introduced in this paper are intended for modeling and analyzing security aspects of dynamic systems including multi-threaded and mobile agent systems. When modeling such systems, additional restrictions may apply with respect to the type of manipulation of the global state that a single ISM can perform, or with respect to the structure of the whole system, e.g., the existence of pre-defined components. Future work will include the provision of d(Amb)ISM frameworks for particular application scenarios and system paradigms, leaving the analyst only with the task of specifying those component ISMs that are specific for the application at hand.

As applications of generic ISMs evolve, new instantiations may turn out to be useful and may be implemented by ourselves and/or third parties.

We plan to extend the proof support offered by our Isabelle implementation of the ISM meta theory, as far as required by applications. Since our aim is to prove security, which typically is a collection of safety (but not liveness) properties, the most important next steps will be proof support for refinement and compositionality, i.e. proof decomposition wrt. parallel composition.

7 Conclusion

We have generalized Interacting State Machines, instantiated them with dynamic features, and demonstrated how to employ them for system modeling.

The extension is based on the introduction of global state for a system of ISMs. While generic ISMs do not further specify the structure of the global state and the type of the commands, different instantiations supporting different kinds of dynamics can be given. We have done so by defining dynamic ISMs allowing for mutable communication interfaces and a form of creation and deletion of components, as well as Ambient ISMs considering mutable contexts of components and components moving between contexts. The latter variant gives an operational semantics to the (boxed) ambient calculus. Furthermore, both variants can be combined to form dynamic Ambient ISMs, thus giving a very expressive and flexible approach to dynamic automata. We have defined generic and dynamic ISMs fully and formally, while the formalization and use of Ambient and dynamic Ambient ISMs is the subject of the companion paper [KO03].

By providing the concept of a global state appearing in different flavors, the ISM approach can be tailored to the analysis task at hand. Each of the extensions presented can be used on its own or be combined depending on the system that is to be analyzed. This is seen as an important practical advantage, since it does not leave users with the burden of additional structure if there is no need to, but on the other hand gives high expressive power where required.

Since we have fully formalized generic ISMs and their descendants in Isabelle/HOL, we both inherit its advanced interactive and semi-automatic proof support features and achieve maximal reliability of the results. Experience already gained with real-world application examples, e.g. [OL02,OWL03] using basic ISMs, indicates that a high degree of proof automation can be achieved in this way, even though typical reactive systems have infinite state space and transitions heavily depend on message buffer contents and other (local and global) data.

Acknowledgments Thomas Kuhn has provided invaluable input motivating the design of the ISM extensions. We thank him as well as some anonymous referees for their suggestions on early versions of this paper.

References

BCC01. Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Boxed ambients. In *TACS 2001, 4th. International Symposium on Theoretical Aspects of Computer Science*, volume 2215 of *LNCS*, pages 38–63. Springer-Verlag, 2001.

CG98. Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computational Structures: First International Conference, FOSSACS '98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.

EHS97. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL: Formal Object-Oriented Language for Communicating Systems.* Prentice Hall, 1997.

GR95. Radu Grosu and Bernhard Rumpe. Concurrent timed port automata. Technical Report TUM-I9533, Technische Univerität München, 1995.

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

HLN+90. David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering*, 16(4):403–414, 1990.

HS97. Ursula Hinkel and Katharina Spies. Spezifikationsmethodik für mobile, dynamische FOCUS-Netze. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch 1997*, 1997.

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.

KO03. Thomas Kuhn and David von Oheimb. Interacting State Machines for mobility. In *Proc. of the* 12th *International FME Symposium (FM'03)*. Springer, September 2003. http://ddvo.net/papers/ISMfM.html, to appear.

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.

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

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/.

Ohe02. David von Oheimb. Interacting State Machines: *a stateful approach to proving security*. In Ali Abdallah, Peter Ryan, and Steve Schneider, editors, *Proceedings from the BCS-FACS International Conference on Formal Aspects of Security 2002*, volume 2629 of *LNCS*, pages 16–33. Springer-Verlag, 2002. http://ddvo.net/papers/ISMs.html.

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* 7th *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/.

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)*. Spinger, 2003. http://ddvo.net/papers/SLE88_MM.html, to appear.

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/.

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

Zap02. Júlia Zappe. Towards a mobile TLA. In *Proc. of the 7th ESSLLI Student Session, 14th European Summer School in Logic, Language and Information, Trento, Italy*, 2002.