Abstract Communication Model for Distributed Systems

Uwe Glässer, Yuri Gurevich, and Margus Veanes

Abstract—In some distributed and mobile communication models, a message disappears in one place and miraculously appears in another. In reality, of course, there are no miracles. A message goes from one network to another; it can be lost or corrupted in the process. Here, we present a realistic but high-level communication model where abstract communicators represent various nets and subnets. The model was originally developed in the process of specifying a particular network architecture, namely, the Universal Plug and Play architecture. But, it is general. Our contention is that every message-based distributed system, properly abstracted, gives rise to a specialization of our abstract communication model. The purpose of the abstract communication model is not to design a new kind of network; rather, it is to discover the common part of all message-based communication networks. The generality of the model has been confirmed by its successful reuse for very different distributed architectures. The model is based on distributed abstract state machines. It is implemented in the specification language AsmL and is used for testing distributed systems.

Index Terms—Abstract state machines, communication protocols, computer networks, distributed systems, requirement specification, system modeling, testing of distributed systems.

1 Introduction

A couple of years ago, the group on Foundations of Software Engineering at Microsoft Research (FSE) and, in particular, the authors of the present paper worked on a high-level model of the Universal Plug and Play (UPnP) architecture [39], [20]. It occurred to us that the UPnP communication model is a specialization of an abstract communication model (ACM) that is so general that every distributed system that involves a communication network gives rise to a specialization of ACM. The simple examples that we could think of confirmed our thesis. Various projects that FSE worked on after that provided additional and more convincing confirmation. One example is an XML-based formal language called XLANG for defining the data and network protocols of automated business processes [44]. XLANG is quite involved and quite different from UPnP and from any previous project that we have been involved with. Nevertheless, the appropriate high-level XLANG communication model was obviously a specialization of the ACM. The largest project that FSE has been involved with is Indigo [13], a unified programming model and communications infrastructure for distributed service-oriented design. Again, Indigo is quite different from (and quite larger than) any prior project that we have been involved with. And, again, the appropriate high-level Indigo communication model was a specialization of the ACM. This experience convinced us that the ACM is sufficiently important to be presented in its own right. That is exactly what we do in this paper.

The ACM is written in a high-level executable specification language AsmL [4] developed by FSE and based on the concept of *abstract state machine* or ASM [25]. AsmL is integrated with Microsoft's software development, documentation, and runtime environments. It compiles to the .NET intermediate language and has full .NET interoperability. AsmL is a practical instrument for systems design (and reverse engineering). Furthermore, FSE is developing a host of testing and validation tools on the AsmL platform.

During the recent years, model-based design, specification, analysis, and testing have been getting increasing attention in the context of industrial software development. Part of the reason for this is the shift from monolithic applications designed to run on a single machine to programming platforms designed to run in a distributed service-oriented environment which have to conform to specific usage protocols. In particular, the upcoming version of Windows will contain the Indigo system. In this vein, we apply the AsmL-based tools to concrete ACM specializations in concrete modeling contexts. This is illustrated below.

Abstract state machines simulate arbitrary algorithms in the step-for-step manner. There is a substantial experimental confirmation [3], [18] as well as theoretical confirmation [26], [11] of that thesis. ASMs have been used to specify various architectures, protocols and languages, in particular, C, Java, SDL, and VHDL [3], [12]. The International Telecommunication Union adopted a comprehensive ASM-based formal definition of SDL [23] as an integral part of the current SDL standard; in the meantime, that definition has been rendered in AsmL [34].

Let us make a few comments on high-level rigorous specifications. Some features of such specifications are well recognized in the academic community as advantageous.

- *U. Glässer is with the School of Computing Science, Faculty of Applied Science, Simon Fraser University, Burnaby, British Columbia, Canada V5A 1S6. E-mail: glaesser@cs.sfu.ca.*
- *M. Veanes and Y. Gurevich are with Microsoft Research, One Microsoft Way, Redmond, WA 98052. E-mail: {gurevich, margus}@microsoft.com.*

Manuscript received 30 May 2003; revised 30 Dec. 2003; accepted 5 Apr. 2004.
Recommended for acceptance by R. Lutz.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0064-0503.

While informal documentation is often ambiguous, incomplete, and even inconsistent, properly constructed formal specifications are consistent, avoid unintended ambiguity, and are complete in the appropriate sense that allows for intended ambiguity (nondeterminism). Let us emphasize though that, in practice, formal specifications build on given informal descriptions. You fix loose ends, resolve unintended ambiguities and inconsistencies, separate concerns, etc. Gradually, the given informal description gives rise to a mathematical model or to a hierarchy of such models.

Some other features of high-level rigorous specifications are more controversial. We advocate AsmL specifications that are executable and written in the style of literate programming so that they are easy to comprehend; see examples in [4]. AsmL is used to explore and validate the requirements and the design, and to test the conformance of the implementation to the specification. Executable specifications can be used for test-case generation [24], runtime verification [5], and scenario-based modeling and testing [7].

And, there are features of high-level rigorous specifications, at least of ASM specifications, that have not been given sufficient attention by the academic community. An ASM model is a closed world with well delineated interfaces to the outside world. The need to make that world closed provokes one to fill various gaps in a given informal spec. That is what happened when we worked on the Universal Plug and Play (UPnP) architecture [20]. While the informal documentation [39] described UPnP devices and the UPnP protocol, it did not provide a conceptual model of the network. We had to construct such a model. It turned out to be quite general and could be further generalized by abstracting from particular protocols. That is how we arrived to the ACM.

We tried to make this paper self-contained. In Section 2, we provide the ASM semantics of a portion of AsmL sufficient for our purposes in this paper. The abstract communication model ACM is described in Section 3. The ACM is illustrated in Section 4, where we describe the relevant part of the UPnP model and, in Section 5, where we describe the relevant part of the XLANG model. In Section 6, we illustrate the use of the abstract communication model for test case generation. In Section 7, we discuss related work.

2 Abstract State Machines and AsmL

Our method rests on the ASM theory. To deploy ASMs in an industrial environment, we need an industrial-strength language. One such language has been (and is being) developed in Microsoft Research. It is called AsmL (ASM Language). Here, we focus on those aspects of AsmL that are most important for the general understanding and that are actually used in this paper. The description given here is incomplete in many respects. For an in-depth introduction to AsmL, we recommend the reader to consult [4]. The interested reader may also want to consult [25], [26], [11] for ASM theory, but we do not presume that the reader is familiar with the ASM theory. We do presume that the reader has some familiarity with the object oriented paradigm.

First, we explain how the fundamental concepts of ASM *states* and *updates* are reflected in AsmL and we explain the semantics of the core programming constructs of AsmL in self-explanatory ASM terms; for a more comprehensive treatment see [29]. Then, we introduce additional functionality into the modeling framework that enables us to faithfully simulate distributed ASM agents; such simulation is needed because the current version of AsmL lacks runtime support for true concurrency.

2.1 States

The state of an AsmL model is given by the values (or interpretations) of the state vocabulary symbols that occur in the model program.[1] The vocabulary symbols are function symbols; each function symbol has a fixed arity. Every vocabulary symbol is a *constant* or a *variable* (i.e., a *dynamic function*). Variables are marked by the keyword `var` and their values are allowed to change from state to state, whereas all constants keep their initial values. (Note that a state of say a C program is not given like that; it has implicit parts, namely, the stack and the program counter.)

All vocabulary symbols are strongly typed. AsmL has a rich type system containing type constructs for sequences, sets, maps, bags, etc. As far as this paper is concerned, all dynamic functions symbols are either nullary or unary. In the latter case, they are either *instance fields of classes or dynamic universes*. Formally, a dynamic universe C is a Boolean-valued unary function; we say that o is in C if $C(o) = true$.

In the following AsmL program, there is a fixed enumerated type `Airport` whose elements are the airports `ARN`, `CPH`, and `SEA`. In ASM terms, `Airport` is a *static universe* where each element is a *static nullary function*. `Airline` is a class; semantically it is a *dynamic universe* that consists, in a given state, of the airline objects created so far; initially the universe is empty. The (constant) field `name` associates with every airline a string; `name` is a *unary function* from airlines to strings. The variable field `flights` associates with every airline a set of airport pairs (a flight table); *flights* is a *dynamic unary function* from airlines to flight tables. `NationalAirline` is a subclass of `Airline`; the dynamic universe of `NationalAirline` is a subset of that of `Airline`. The global variable `airlines` is a set of airlines that is initially empty; `airlines` is a *dynamic nullary function*. The global variable `myAirline` is either of type `Airline` or has the special *null* value (indicated by the question mark) that is also its initial value.

```
enum Airport
 ARN
 CPH
 SEA

type Flights = Set of(Airport,Airport)

class Airline
 name as String
 var flights as Flights
```

1. In the terminology of mathematical logic states are *first-order structures*.

```
class NationalAriline extends Airline

var airlines as Set of Airline = {}

var myAirline as Airline? = null
```

2.2 Updates

We view a state as a kind of memory. A *location* of a state is either a nullary variable f or a pair (f, o) consisting of a unary dynamic function f and an object o of the right type. We say that a location f or (f, o) is variable if f is dynamic. The *content* or *value* of a location f or (f, o) in a given state A is the value of f or $f(o)$ in A, denoted by f^A or $f^A(o)$, respectively. An *update* is a pair $l \to a$, where l is a variable location and a is a value of the type appropriate for l. To *fire* this update in a state, replace the current content of l with a.

The updates explained above are called *total*. In addition, AsmL supports *partial updates*. In the most common case, partial updates are used to allow simultaneous pointwise modifications of sets and maps. We explain partial updates of a variable location l of a type set of T. A partial update of l is a request to *add* or *remove* an element a of type T, denoted here by $l \to^+ a$ or $l \to^- a$, respectively. Firing a partial update has the obvious meaning.[2] The theory of partial updates is developed in [27], [28]. By an *update set*, we mean a set of total and partial updates. An update set U is *consistent* if, for each location l in U, there is either a single total update to l, or all updates to l are partial and there are no two partial updates $l \to^+ a$ and $l \to^- b$ where $a = b$. To *fire* a consistent update set U in a state A, fire all the updates in U simultaneously.

2.3 Programs

We formally define here the semantics of the main AsmL program constructs including those used in this paper. We omit numerous details.[3] The basic building blocks of programs are *statements* and *expressions*. A statement or an expression is evaluated in a given state. In the usual programming languages, the state may change during an evaluation. This is not the case with AsmL. The state does not change during the evaluation of the program. The whole program describes one step.

The *value semantics* of an expression E in a state A is the value of E in A, denoted by E^A. The semantics of expressions is given by a straightforward induction on the structure of expressions and is presented only partially here. (Some expression evaluations produce side effects in the form of updates.) The most important case is that of *import expressions*:

- To evaluate an expression `new` $C(E_1, \ldots, E_k)$ of class C with k uninitialized fields $f_1, \ldots, f_k$, produce a new object o of type C such that each $f_i^A(o) = E_i^A$. This evaluation has a side effect of expanding the dynamic universe of C, and any dynamic universe it is a subset of, with o.

2. This is a simplified version of partial updates of a set location that is adequate for the purposes of this paper.

3. The exposition here is simplified in many ways; in particular, we do not take into account exception handling since we do not use it in this paper.

The *update semantics* of a statement R in a state A is the set of all updates produced by evaluating R in A (including the side effects of expression evaluations). We proceed by case analysis over statements. We use o, a for elements of the universe, x, y, v for variables, C for class identifiers, D, E for expressions, and S, R for statements. All expressions and statements are assumed to be type correct.

2.3.1 Basic Programs

- To evaluate the statement `skip` do nothing. In other words, no updates are produced.
- To evaluate an *assignment* `v :=`E, produce the total update `v` $\to E^A$. To evaluate an *assignment* D`.v :=`E, produce the total update $(\texttt{v}, D^A) \to E^A$.
- To evaluate a statement `add` E `to` v, produce the partial update $v \to^+ E^A$. To evaluate a statement `remove` E `from` v, produce the partial update $v \to^- E^A$. The cases of `add` E `to` $D.v$ and `remove` E `from` $D.v$ are similar.
- To evaluate a statement `if` E `then` S `else` R, examine the value E^A. If it is *true*, evaluate S, otherwise evaluate R. The else part is optional and if omitted corresponds to `elseskip`.
- To evaluate a *do-in-parallel block* of statements

$$\begin{array}{l} R_1 \\ R_2 \\ \ldots \\ R_3 \end{array}$$

  evaluate all the statements R_i simultaneously.

As an example, let A be the initial state for the sample state declaration above and let P_1 be the following program:

```
myAirline := new Airline("SAS",{})
if myAirline = null then skip else...
```

This is a do-in-parallel block of two statements. The import expression returns a new airline object o such that $\texttt{name}^A(o) = \text{"}SAS\text{"}$ and $\texttt{flights}^A(o)$ is the empty set and produces the side effect $(\texttt{Airline}, o) \to true$. The assignment produces $\texttt{myAirline} \to o$ and $(\texttt{Airline}, o) \to true$, which are also the only updates produced by P_1 because $\texttt{myAirlines}^A = null$.

- To evaluate a statement

$$\begin{array}{l} \texttt{let } x = E \\ R(x) \end{array}$$

  evaluate $R(E^A)$.

To illustrate the use of a let-statement consider the following program P_2:

```
let x = new Airline("SAS",{})
 myAirline := x
 x.flights := {(CPH,SEA),(SEA,CPH)}
```

As in P_1, an airline object o with name $\text{"}SAS\text{"}$ and no flights is created. The evaluation of P_2 produces the total updates

$$\texttt{myAirline} \to o, (\texttt{flights}, o) \to \{(\texttt{CPH}, \texttt{SEA}), (\texttt{SEA}, \texttt{CPH})\}$$

and $(\texttt{Airline}, o) \to true$.

ASMs given by basic programs are sequential algorithms; in particular, they are nondistributed algorithms with uniformly bounded parallelism. The latter means that the number of actions performed in parallel is bounded independently of the state or the input. The notion of sequential algorithms is formalized in [26] where it is proven that, for every sequential algorithm, there is a basic ASM that simulates the algorithm step for step.

2.3.2 Parallel Programs

The AsmL type system includes the background library of collection types such as sets, maps, bags (multisets), and sequences and allows one to construct collections dynamically by *comprehension*. For example, the comprehension expression {x | x in [1..10] where x mod 2 = 0} produces the set of even integers between 1 and 10. The library also includes all the standard operations for the respective collection types. In theory, the set background alone is sufficient [11].

The construct that distinguishes parallel programs from basic programs is the *forall* construct.

- To evaluate

  forall x in E
  $R(x)$

  in a given state A, evaluate $R(a)$ in A simultaneously for every element a in the collection E^A.

In most common cases, the forall-statement is used in combination with partial updates. Consider the program P_3:

```
let x = new Airline("SAS",{})
 myAirLine := x
 forall a in {ARN,SEA}
  add (CPH,a) to x.flights
```

P_3 produces total updates $(\texttt{Airline}, o) \rightarrow \mathit{true}$ and $\texttt{myAirline} \rightarrow o$, where o is a new object as in P_1. In addition, it produces the partial updates $(\texttt{flights}, o) \rightarrow^+ (\texttt{CPH}, \texttt{ARN})$ and $(\texttt{flights}, o) \rightarrow^+ (\texttt{CPH}, \texttt{SEA})$.

The appropriate notion of parallel algorithms is formalized in [11] where it is proved that, for every parallel algorithm, there is a parallel ASM that simulates the given algorithm step for step.

2.3.3 Nondeterministic Programs

A modeling language should allow us to abstract from irrelevant details of the system behavior. One of the most common abstractions is to allow multiple possible behaviors in the model. To this end, AsmL has a *choose construct* for expressing explicit nondeterminism.

- To evaluate

  choose x in E where $D(x)$
  $R(x)$
  ifnone R'

  choose nondeterministically an element a from the collection E^A such that $D^A(a) = \mathit{true}$ and evaluate $R(a)$. If no such element exists, evaluate R'. The where part is optional and if omitted corresponds to where true. Also, the ifnone part is optional and if omitted corresponds to infnone skip.

The meaning of a nondeterministic program P is the set of *all possible* evaluations of P which is finite since all collections in AsmL are finite.

For example, let A be a state where

$$\texttt{flights}(\texttt{myAirline})^A = \{(\texttt{CPH}, \texttt{ARN}), (\texttt{CPH}, \texttt{SEA}), (\texttt{SEA}, \texttt{CPH})\}$$

and imagine that you want to remove one of the flights, never mind which, departing from CPH. The following program P_4 does that.

```
choose x in myAirline.flights where
      First(x) = CPH
 remove x from myAirline.flights
```

This program has two possible evaluations, one that produces the partial update $l \rightarrow^- (\texttt{CPH}, \texttt{ARN})$ and the other that produces the partial update $l \rightarrow^- (\texttt{CPH}, \texttt{SEA})$, where $l = (\texttt{flights}, \texttt{myAirline}^A)$. When you run this AsmL program, only one of the evaluations is fired. However, there is an advanced AsmL construct explore E (not used in this paper) that goes through all the possible evaluations of a given expression and collects various data. This construct is particularly important in the context of developing tools such as the model explorer mentioned in Section 6.

2.3.4 Methods

As usual, AsmL has methods in addition to fields. Parameters to methods are passed by value.

- To evaluate $m(E_1, \ldots, E_k)$ in A where m is a method name and the definition (body) of m is a statement $S(v_1, \ldots, v_k)$, evaluate $S(E_1^A, \ldots, E_k^A)$ in A. The evaluation of an instance method call $D.m(E_1, \ldots, E_k)$ is similar, where D^A is substituted for the keyword me in the body of m.

A method call may return a value. This motivates the *return statement* construct.

- To evaluate

  S
  return E

  evaluate S and return the value of E^A. This presumes some syntactic restrictions on S. In particular, S cannot contain a return statement. If S is skip, S can be omitted.

By *firing* a method $m(E_1, \ldots, E_k)$ in a state A, we mean first evaluating $m(E_1, \ldots, E_k)$ in A and then firing the resulting updates in A.

In the following example, the method FlipCoin takes no arguments and returns (nondeterministically) either "heads" or "tails". The method ChooseSubset takes a set of elements and returns a random subset of this set.

```
FlipCoin() as String
 choose x in {"heads","tails"}
  return x

ChooseSubset(elems as Set of Object)
```

```
    as Set of Object
 return
    {e | e in elems where FlipCoin() = "heads"}
```

Thus, expression evaluation (and not only statement evaluation) may be nondeterministic. For example, there are $2^{1,000}$ possible evaluations of the program

```
x := ChooseSubset({1..1000})
```

2.4 Distributed ASMs and Simulation of Agents in AsmL

Until now, we dealt with one-agent ASMs. A *distributed ASM*(DASM) involves a collection of *agents* that perform their computation steps concurrently. From the global point of view, agents are elements of a dynamic universe *Agent* that may grow and shrink over a DASM run. Ideally, one would like to have a distributed runtime environment for executing distributed ASMs. The current distribution of AsmL does not yet have runtime support for true concurrency; this is a work in progress. For the time being, we simulate concurrent behavior by means of interleaving as explained below. The reader interested in the semantics of distributed ASMs is referred to [25].

Agents are viewed as objects of a class `Agent`. The program of an agent a is a method of the class `Agent`. The state of a (given by all fields of a) evolves in sequential steps with each invocation of its program.

In this paper, every agent a has a field `mailbox` and a method `InsertMessage` that is used by other agents to send messages to a. The agent a also has a method `IsActive` that determines whether the agent is active in the current state that is *true* by default. (In general, ASM agents are not required to have mailboxes.)

```
type MESSAGE
class Agent
 abstract Program()
 virtual IsActive() as Boolean
   return true
 var mailbox as Set of MESSAGE = {}
 InsertMessage(m as MESSAGE)
   add m to me.mailbox
```

Several agents may simultaneously insert messages into the mailbox of a. This will not cause a conflict in updating the mailbox because these updates are partial.

Recall the method `ChooseSubset` introduced above. For simulation purposes, a distributed program has a global method `RunAgents`. By firing `RunAgents`, we perform a single step of the top-level system. Thus, at each global step of the system, some, none, or all of the active agents in the system may perform a step.

```
RunAgents()
 forall a in ChooseSubset({a | a in Agent where
    a.IsActive()})
  a.Program()
```

Remark. A perceptive reader may notice a problem with this method. If there is a shared memory, then two active agents may do contradictory things, for example, one of them may write 7 and another one may write 11 into a shared location. In such a case, an exception will be raised. Alternatively, `ChooseSubset` can be rewritten to avoid possible contradictions.

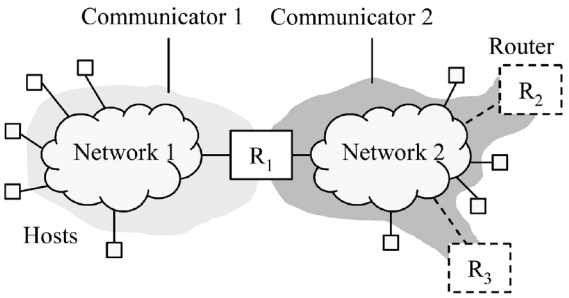


Fig. 1. An instance of the abstract communication model in which Communicator 2 abstractly models the three routers R_1, R_2, and R_3 in one direction, respectively.

3 Abstract Communication Model

One may wonder how to deal with networks in a sufficiently abstract and general way. Often, one wants to clearly separate the behavior of the network from the behavior of an application running over this network. This problem came up in a number of our projects, initially in the UPnP project. To solve this problem, we introduced a special category of agents which we call *communicators*. Each communicator abstractly represents a multitude of routers in a TCP/IP network [15]. Intuitively, different communicators represent disjoint subnets (see Fig. 1) and, typically, this is indeed the case. But, we do not impose this restriction. The communicators transfer messages between applications. We emphasize that, even though the model was obtained by abstraction from TCP/IP networks, it is independent from TCP/IP and is used to deal with very different networks.

```
class COMMUNICATOR extends Agent
```

The communicators transfer messages between *applications* running on hosts connected to the network. Thus, one can think of communicators as an abstract kind of "router" of messages. However, the term "router" used in this sense is much more general than the corresponding TCP/IP term.

```
class APPLICATION extends Agent
```

Assuming an open system view, a system consisting of applications and communicators is affected in various ways by the operational environment into which it is embedded. For instance, external actions and events may change the system configuration, affect the communication load of the network, or otherwise have an impact on message delays. Those external actions and events are basically unpredictable as far as the system is concerned. In other words, nondeterministic behavior plays an important role.

Communicators are nondeterministic in two ways. There is an internal nondeterminism that reflects abstraction of various details of routing. In addition, there is environment-induced nondeterminism that may cause, e.g., that some messages get lost. For example, in the case of UPnP, one uses the TCP protocol, which is reliable, and the UDP

protocol, which is not reliable. It is the responsibility of an application to tolerate the nonreliable behavior of the network.

In the following sections, we first discuss the various aspects of the communicator's operation and then present the main program that integrates these aspects.

3.1 Message Transformation

Not all messages have a single recipient. Some messages are intended to be sent to a group of recipients. Though, multicasting is just one example of a general class of transformation operations for message processing. Other transformations, for instance, include incrementing a hop count for time-to-live calculations, corruption and encryption of messages.

The `ResolveMessage` method transforms a message (an inbound message of the communicator) into a set of messages (outbound messages of the communicator). For example, the transformation may involve adding or removing header information or converting a multicast message into an equivalent set of unicast messages. The model places no restriction on the kind of transformation performed in this step. It is even possible that a transformation may discard a message completely, by returning an empty set. By default, the message is left untransformed.

```
class COMMUNICATOR
 virtual ResolveMessage(m as MESSAGE) as
      Set of MESSAGE
   return {m}
```

In the TCP/IP world, addressing mechanisms classify as *unicasting*, *broadcasting*, or *multicasting*, where multicasting can be viewed as the most general one [15]. In our model, a multicast can involve any set of applications that are reachable over the network; in principle, every such set of applications may have an address. The receiver addresses of a multicast message can themselves be multicast addresses. An *address table* of a communicator is a (possibly dynamic) mapping whose domain consists of the addresses a of some multicast groups that the communicator can deal with. If a is the address of a multicast group g, then `addressTable`(a) is a set that consists of the addresses of some multicast subgroups of g which could be singleton groups. The union of all the subgroups is g itself.

```
type ADDRESS
class COMMUNICATOR
 var addressTable as Map of ADDRESS to
     Set of ADDRESS
```

The address table is used in the process of resolving an inbound message into a set of transformed outbound messages.

```
class COMMUNICATOR
 abstract Destination(m as MESSAGE) as ADDRESS
 virtual Transform(m as MESSAGE, dest as
     ADDRESS) as MESSAGE
  return m
 ResolveMessage(m as MESSAGE) as Set of MESSAGE
  return{me.Transform(m,a) | a in
    me.addressTable(me.Destination(m))}
```

3.2 Message Routing

Communicators determine the recipient of a message. Presumably, this is done by examining addressing information in the headers and reconciling that information with the communicator's knowledge of network topology.

The `Recipient` method of a communicator identifies which agent ought to receive an outbound message. The recipient may be an application running on a local host, one that is connected directly to the communicator, or another communicator that will forward the message further. The message may also have no recipient in which case the return value of the method is *null*; this possibility forces us to use the type `Agent?` consisting of agents and the *null* value.

```
class COMMUNICATOR
 abstract Recipient(m as MESSAGE) as Agent?
```

A generic way to encode global network topology, as far as this information is required in an abstract communication model (choosing a degree of detail and precision as needed), is through a (possibly dynamic) mapping called `routingTable`. The routing table of a communicator is supposed to map addresses to neighboring agents (communicators or applications) as required for routing messages through a network.

```
class COMMUNICATOR
 var routingTable as Map of ADDRESS to Agent
```

The recipient of an outbound message is determined by looking up the destination address of the message in the routing table.

```
class COMMUNICATOR
 Recipient(m as MESSAGE) as Agent?
  let addr = me.Destination(m)
   if addr in me.routingTable then
     return me.routingTable(addr)
   else return null
```

3.3 Delivery Conditions

In real-world distributed systems, there are complex conditions that govern when (or if) a message is forwarded by a communicator. These might include network latency, security parameters, and resource limitations of the underlying physical network. Since we abstract here from lower-level network layers, the decision whether a set of messages are ready to be delivered in a given state of the network is abstractly stated through a function `ReadyToDeliver` effectively serving as an oracle.[4]

```
class COMMUNICATOR
 abstract ReadyToDeliver(messages as Set
   of as MESSAGE) as Set of MESSAGE
```

Note that messages that are never ready to deliver are in effect "lost," even though they persist in the communicator's

4. Clearly, one has to provide some kind of implementation for the function `ReadyToDeliver` in order to make the model executable; nonetheless, any such details indeed depend on the operational environment of the system and as such remain invisible within the system.

mailbox. For example, some UDP message m may never be ready to be delivered.[5]

3.4 Message Delivery

We are now ready to present an algorithm for how communicators route messages through a network. A communicator's control program forwards messages found in its mailbox by inserting them into the mailboxes of the respective recipients of the message. Notice that a communicator program is nondeterministic if `ReadyToDeliver` is so.

```
class COMMUNICATOR
 override Program()
  forall msg in me.ReadyToDeliver(me.mailbox)
   remove msg from me.mailbox
              //delete the message
   forall m in me.ResolveMessage(msg)
              //consider all resolved messages
    let a = me.Recipient(m)
     if a ≠ null then
              // if recipient found
      a.InsertMessage(m)
              // forward the message
```

First, the communicator determines the subset of unprocessed messages that are ready to be delivered. Note that some, all, or none of the messages in the mailbox may be selected for processing. Next, the communicator transforms each selected message, as described above. This may result in the unfolding of a single message into many messages, each of which will be posted to a single recipient (for instance, in the case of multicasting). Note that a recipient may be another communicator. Finally, the communicator calculates the recipient of each resolved message and inserts the (transformed) message to the mailbox of the recipient.

4 Universal Plug and Play

The *Universal Plug and Play Architecture* (UPnP) [39] is an industrial standard for dynamic peer-to-peer networking defined by the UPnP Forum [40]:

> Universal Plug and Play is a distributed, open networking architecture that leverages TCP/IP and the Web technologies to enable seamless proximity networking in addition to control and data transfer among networked devices in the home, office, and public spaces.

We have developed a high-level executable DASM model of the UPnP architecture [20], [21], [22] based on the informal requirements specification [39]. The construction of a DASM allows us to combine both *synchronous* execution models and *asynchronous* execution models within one uniform model of computation. Thus, we model an ensemble of devices and control points and the communication network in between them through the composition of two separate DASMs, one for the communication endpoints and one for the network. We preserve the synchronous nature of devices and control points by mapping them onto DASM agents where each individual agent on its own realizes a synchronous execution model. Here, we give an overview of our UPnP model focusing on interoperability aspects (rather than on the internal behavior of UPnP components) related to the abstract communication model.

5. This behavior reflects the fact that UDP's best-effort packet delivery mechanism sometimes fails to deliver a message depending on conditions and events that remain invisible for the user.

4.1 The UPnP Protocol

We briefly summarize here the basic characteristics of the UPnP protocol. Technically, it is a layered protocol built on top of TCP/IP by combining various standard protocols including DHCP, SSDP, SOAP, and GENA. It supports dynamic configuration of any number of *devices* offering various kinds of *services* requested by *control points*. To perform control tasks, a control point needs to know what devices are available (i.e., reachable over the network), what are the services advertised by devices, and when those advertisements expire. The services of a device interact with the external (physical) world through the actuators and sensors of the device.

The UPnP protocol defines six basic steps or phases. Initially, these steps are invoked one after the other in the order given below, but may arbitrarily overlap afterward.

0. *Addressing* is needed for obtaining an IP address when a new device is added to a network.
1. *Discovery* informs control points about the availability of devices and their services.
2. *Description* allows control points to retrieve detailed information about a device and its capabilities.
3. *Control* provides mechanisms for control points to access and control devices through well-defined interfaces.
4. *Eventing* allows control points to receive information about changes in the state of a service at runtime.
5. *Presentation* enables users to retrieve additional device vendor specific information.

Control points and devices interact through exchange of messages over a TCP/IP network where network characteristics, like bandwidth, dimension, and reliability, are left unspecified. In general, the following restrictions apply. Communication is considered to be neither predictable nor reliable; that is, message transfer is subject to arbitrary and varying delays, and some messages may never arrive. Devices may appear and disappear at any time with or without prior notice. Consequently, there is no guarantee that a requested service is available in a given state or will become available in future. In particular, an available service may not remain available until a certain control task using this service has been completed.

4.2 UPnP Abstract Machine

The individual communication endpoints, or applications, in UPnP are *devices* and *control points*.

```
class CONTROLPOINT extends APPLICATION
class DEVICE       extends APPLICATION
```

In addition to communicators and applications, the full model employs some additional agents that reflect well identified parts of the external world, e.g., DHCP server agents, but here we ignore them. With each agent type, we

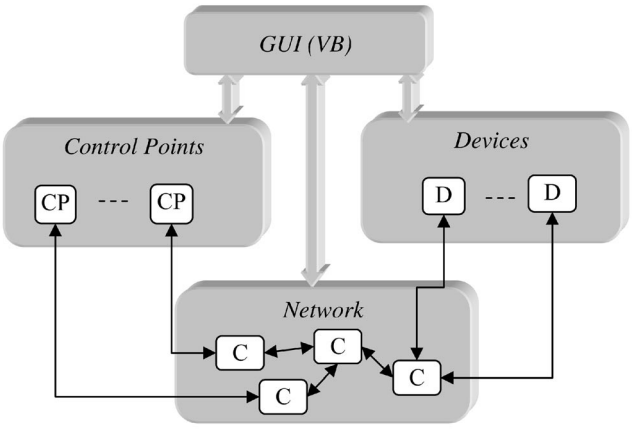


Fig. 2. Instance of UPnP abstract machine.

associate one or more interfaces for interaction with other agents in our model or with the environment that is the external world. The environment affects the system behavior in various ways. For example, it changes the system configuration, e.g., by creating and removing agents. It also affects the communication load of the network which affects message delays. Those external environmental actions and events are unpredictable.

Our model is integrated with a graphical user interface (GUI) implemented in Visual Basic (VB). This GUI serves for user-controlled interaction and visualization of simulation runs. The overall organization of the model is illustrated in Fig. 2.

4.2.1 Device Model

The purpose of the device model specification is to describe how a device behaves in UPnP-compliant way. In a given system state, a UPnP device may or may not be connected to a network. The network connectivity of a device is affected by actions and events in the external world and may therefore change any time with or without prior notice. The device model specification is a synchronous parallel composition of a number of rules operating in parallel; different rules describe different protocol phases.

```
class DEVICE
 abstract IsConnected() as Boolean
 Program()
  if me.IsConnected() then
    me.RunAddressing()
    me.RunDiscovery()
    me.RunDescription()
    me.RunControl()
    me.RunEventing()
    me.RunPresentation()
```

Here, we focus only on one of those phases, the *control phase* given by *RunControl*, that involves direct interaction with communicators. Every device offers a set of services which can be called by one or more control points by means of messages. Each service call produces a response message sent back to the caller; the response message tells the caller whether the call succeeded or not and may include some return value.

The communication between the devices and the control points is enabled by communicators. Every device is associated with one communicator. A device sends messages by inserting them into the mailbox of that communicator. Typically, a device receives messages from that same communicator (but this is not required: the communicator may represent only a part of the relevant subnet behavior). Notice that who can deliver messages to whom depends on the actual address tables and routing tables used by communicators.

```
type SERVICE
class DEVICE
 services as Set of SERVICE
 var communicator as COMMUNICATOR
 abstract Call(s as SERVICE, msg as MESSAGE)
   as MESSAGE
```

The control phase of the protocol is executed only if the device has a valid address. Initially, the address is *null* but it is eventually updated by the addressing phase of the protocol. When active, the control phase handles service requests one at a time and runs the services.

```
class DEVICE
 abstract IsServiceRequest(m as MESSAGE, s as
    SERVICE) as Boolean
 abstract RunServices()

 var address as ADDRESS? = null
 RunControl()
  if address ≠ null then
   me.RunServices()
   choose msg in me.mailbox
    remove msg from me.mailbox
    choose s in me.services
       where me.IsServiceRequest(msg,s)
     let reply = me.Call(s,msg)
      communicator.InsertMessage(reply)
```

Note that `RunServices()` is executed in parallel to the handling of service requests. After a service call to the device is taken care of, the service may continue to run. Whether only some or all of the services are allowed to run simultaneously depends on the definition of the `RunServices` method of the particular device.

5 Modeling Automated Business Processes

In this section, we summarize a real-life application of distributed abstract state machines to model automated business processes [43]. The main purpose of the summary is to illustrate the effective reuse of the abstract communication model. This very same model is also used in another related application to modeling automated business processes [17], [16].

A *business process* is a protocol for commercial transactions that occur between two or more parties. Transactions are exchanges of goods, services, or information. A typical example of a business process is the series of interactions required to settle a securities trade. Many business processes are designed to span organizational boundaries.

For example, a process for corporate purchasing may include roles for a "buyer," a "seller," and a "shipping agent, " where each of the parties is a separate enterprise.

An *automated business process* is executed without manual steps. For example, banks in the United States use an automated clearinghouse to settle accounts for checks they honor on each other's behalf. A protocol for an automated business process specifies data formats for messages, some constraints on the behavior of the electronic communications network itself, and a description of the possible patterns of messages that constitute a transaction.

5.1 XLANG

XLANG is an XML-based formal language that can be used to define the data and networking protocols of automated business processes [44]. XLANG builds on the existing standards for the Internet and World Wide Web. The building block standard that XLANG is most dependent on is WSDL, the Web Service Description Language [42]. XLANG has a two-fold relationship with WSDL. Syntactically, an XLANG service description is a WSDL service description with an extension that describes the behavior of the service as a part of a business process. Operationally, an XLANG service behavior may rely on simple WSDL services to provide basic functionality for the implementation of the business process.

The goal of XLANG is to make it possible to formally specify business processes as *stateful* long-running interactions. As a rule, business processes involve more than one participant. The full description of a process, called a *contract*, must constraint not only the behavior of each participant, but also the way these behaviors match to comply with the overall process.

5.2 XLANG Abstract Machine

The definition of an abstract operational semantics for XLANG comes in the form of an XLANG abstract machine model formalizing the dynamic properties of the language in terms of machine runs.

An XLANG contract contains two parts:

- a collection of individual so-called *XLANG service behaviors* (that is service behavior specifications) and
- a *port map* defining the interconnection topology of those services.

The port map determines the routing information that is used by the network abstract machine to interconnect the services.

The full XLANG abstract machine is a DASM that has two main components, each of which is again a DASM:

1. *Service Abstract Machine*: A service abstract machine is parameterized with a sequence of XAM instructions.
2. *Network Abstract Machine*: Here, the port map of the contract determines the necessary interconnection topology of the services.

In the remainder of this section, we first outline the overall structure of the service abstract machine. We omit the details regarding the behaviors of the individual XAM instructions. We then describe the interaction of the service abstract machine with the network abstract machine.

5.2.1 Service Abstract Machine

The Service Abstract Machine models an individual service. It consists of two different types of ASM agents: 1) a uniquely identified service *manager* that represents the behavior of the infrastructure on top of which the service runs; 2) some, possibly empty, collection of concurrently operating *processes* (or *process agents*) that represent the XLANG processes associated with that service. Each process represents either a *service instance* created directly by the manager or a subprocess spawned by a previously created process.

During its lifetime, a service instance may spawn several subprocesses. The behavior of that agent group, consisting of the service instance and all its descendants, plays an important role. Each process belongs to some manager. There are four modes that indicate whether

1. a process has exited by having run all of the XAM instructions,
2. a process has been interrupted by an exception,
3. a process has been halted by external intervention, or
4. a process is currently executing instructions.

```
type Service
type Label
type ServiceProgram

class Manager extends Agent
 service as Service
 pgm as ServiceProgram

class Process extends Agent
 manager as Manager
 enum Mode
  exited
  raised
  halted
  running
 var pc as Label
 var mode as Mode
 var subProcesses as Set of Process

class ServiceInstance extends Process
```

The program of a process is to execute the next XAM instruction in the running mode, and to do nothing (skip) in any other mode. Some instructions may be executed without incrementing the program counter; others cause the program counter to jump to a new position in the program.

```
type Instruction
//returns the instruction identified by the
//given label in the given program
Instr(pgm as ServiceProgram, label as Label)
  as Instruction

class Process
 Execute(instr as Instruction)
 Program()
```

```
  if me.mode = running then
   me.Execute(Instr(manager.pgm, me.pc))
```

A manager has two independent jobs. One is to activate new service instances when so called *activating* messages are received. For example, a buyer may send a purchase request to a seller that will trigger the creation of a new instance of the seller service to handle that request. The seller may of course receive several requests from different buyers and create several independent service instances to handle those requests. The other job of the manager is to handle message traffic.

```
class Manager
 Program()
  me.ActivateServiceInstance()
  me.HandleMessageTraffic()
 HandleMessageTraffic()
  me.ReceiveIncomingMessages()
  me.ForwardOutgoingMessages()
```

5.2.2 Interaction with the Network Abstract Machine

The network abstract machine part of the XLANG model is a specialization of the abstract communication model with appropriate routing tables and address tables that enable communication between services whose ports are interconnected according to the port map of the contract.

A service manager has a set of communication *ports*. Each port is associated with an *inbox* and an *outbox* of message instances. The inbox of a port contains all the message instances that have been sent to that port and the outbox contains all the outbound message instances from that port. (The message instance terminology is due to WSDL.)

```
type MessageInst
class Port
 var inbox as Set of MessageInst
 var outbox as Set of MessageInst
class Manager
 ports as Set of Port
```

A message instance is transformed into some concrete *network message* format when it is transmitted over the net. The network message contains the original message instance and a destination port. Here, we identify the `MESSAGE` type introduced above with network messages.

```
type MESSAGE = NetworkMessage
class NetworkMessage
 port as PortName
 msg as MessageInst
```

Each port is associated with a communicator and may be owned by a manager (the manager, if any, who has the port among its ports). No port can be owned by more than one manager (this is enforced by the XLANG language definition [44]).

```
class Port
 communicator as COMMUNICATOR
```

A manager uses the port map of the contract to create network messages from outbound message instances and forwards them to the communicators of the corresponding ports.

```
class Manager
 portMap as Map of Port to Port
class Manager
 ForwardOutgoingMessages()
  forall p in me.ports
   choose m in p.outbox
    remove m from p.outbox
    let msg = new
        NetworkMessage(me.portMap(p),m)
     p.communicator.InsertMessage(msg)
```

When a network message has arrived, the original message instance is extracted from it and inserted into the inbox of the destination port.

```
class Manager
 ReceiveIncomingMessages()
  choose m in me.mailbox
   remove m from me.mailbox
   let p = m.port
    add m.msg to p.inbox
```

Fig. 3 shows an instance of the XLANG abstract machine.

The XLANG model instance in Fig. 3 contains several service abstract machines (SAMs) and a network abstract machine (consisting of several communicators). Each SAM contains a manager, some service instances, and other processes. The XLANG abstract machine has been implemented in AsmL [43]; a GUI is used to control with executions of the model interactively and visualize the machine states during simulation runs.

6 Applications to Model-Based Testing and Analysis

The original purpose in creating the abstract communication model was to reflect the common part of all TCP/IP communication networks. The model allows one to conceptualize complicated networks without using a miraculous transfer of messages from one endpoint to another. The model is useful in high-level modeling and simulation of distributed systems. In the preceding sections, we presented two examples illustrating these points.

It turned out that the model has additional important uses, namely, testing and analysis of distributed systems. Taking into account the space limitation, we illustrate this new point on a minimal example that nevertheless reflects some key aspects of real applications of our technology at Microsoft. We are using here the AsmL tester tool [8] that we refer to below as the *model explorer*.

6.1 Sender-Responder Model

Consider a system of two endpoint agents, a *sender* and a *responder*, connected via two communicators as illustrated in Fig. 4.

For either i, the communicator C_i does not lose messages, and if its mailbox is nonempty, then one of the messages is ready for delivery. The main method of C_i, called Deliver,

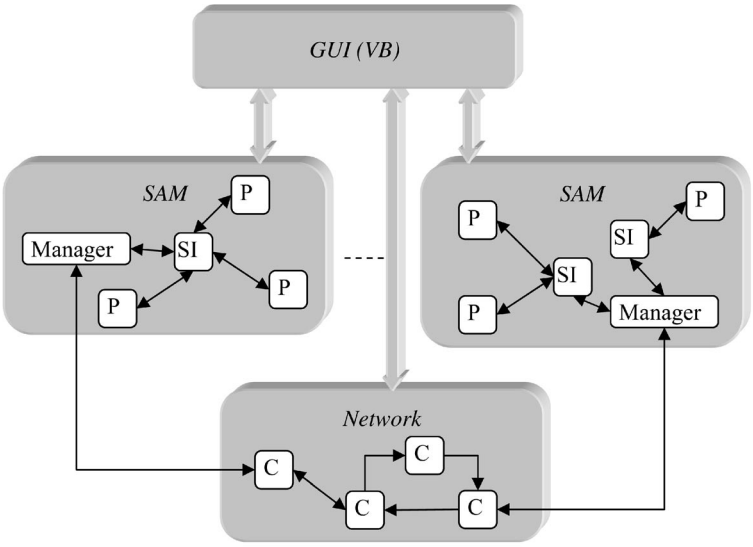


Fig. 3. Instance of XLANG abstract machine.

calls the underlying Program method of the communicator. The sender agent has two methods, Send and Read, and the responder has a single Respond method. There is an Initilalize method that creates the agents and establishes the desired topology shown in the diagram. For simplicity, messages here are represented by strings.

```
type MESSAGE = String

class MyCOMMUNICATOR extends COMMUNICATOR
 name as String
 override ReadyToDeliver(messages as Set of
    MESSAGE) as Set of MESSAGE
  choose m in messages
   return{m}
  ifnone return{}
 Deliver()
  me.Program()

class ENDPOINT extends Agent
 name as String
 var target as Agent? = null

class SENDER extends ENDPOINT
 Send(m as MESSAGE)
  target.InsertMessage(m)
 Read()as MESSAGE
  choose m in me.mailbox
   remove m from me.mailbox
    return m

class RESPONDER extends ENDPOINT
 Respond()
  choose m in me.mailbox
   remove m from me.mailbox
   target.InsertMessage("Re:"+ m)

var initialized as Boolean = false
Initialize()
 initialized := true
 let C1 = new MyCOMMUNICATOR("C1")
 let C2 = new MyCOMMUNICATOR("C2")
 let S = new SENDER("S")
 let R = new RESPONDER("R")
  S.target := C1
  R.target := C2
  ... //configure the communicators
```

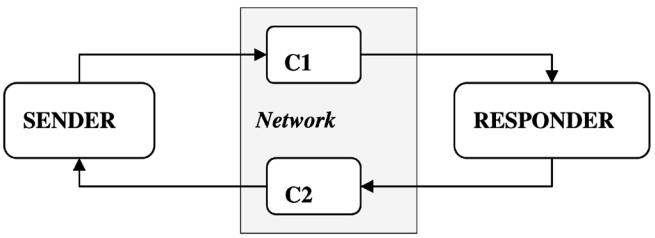


Fig. 4. A system of two endpoint agents, a *sender* and a *responder*.

6.2 Exploring the Sender-Responder Model

Some methods in the model are identified in the model explorer as *actions*. In this particular case, the actions are `Initialize`, `Send`, `Read`, `Respond`, and `Deliver`. In addition, *enabling conditions* are associated with the actions: to `Initialize`, the system must be uninitialized; to `Read`, `Respond`, or `Deliver`, the corresponding agent's mailbox must be nonempty; to `Send(m)`, sender's target mailbox must not contain the message `m`.

The model explorer is used to generate a finite state machine that describes part of the overall behavior of the system. Besides enabling conditions, there are several additional ways in which the user may limit the scope of exploration to avoid state space explosion or to concentrate on particular parts of the state space [24]. In the end, the states of the FSM are the states of the explored part of the system and the transitions are labeled by the appropriate

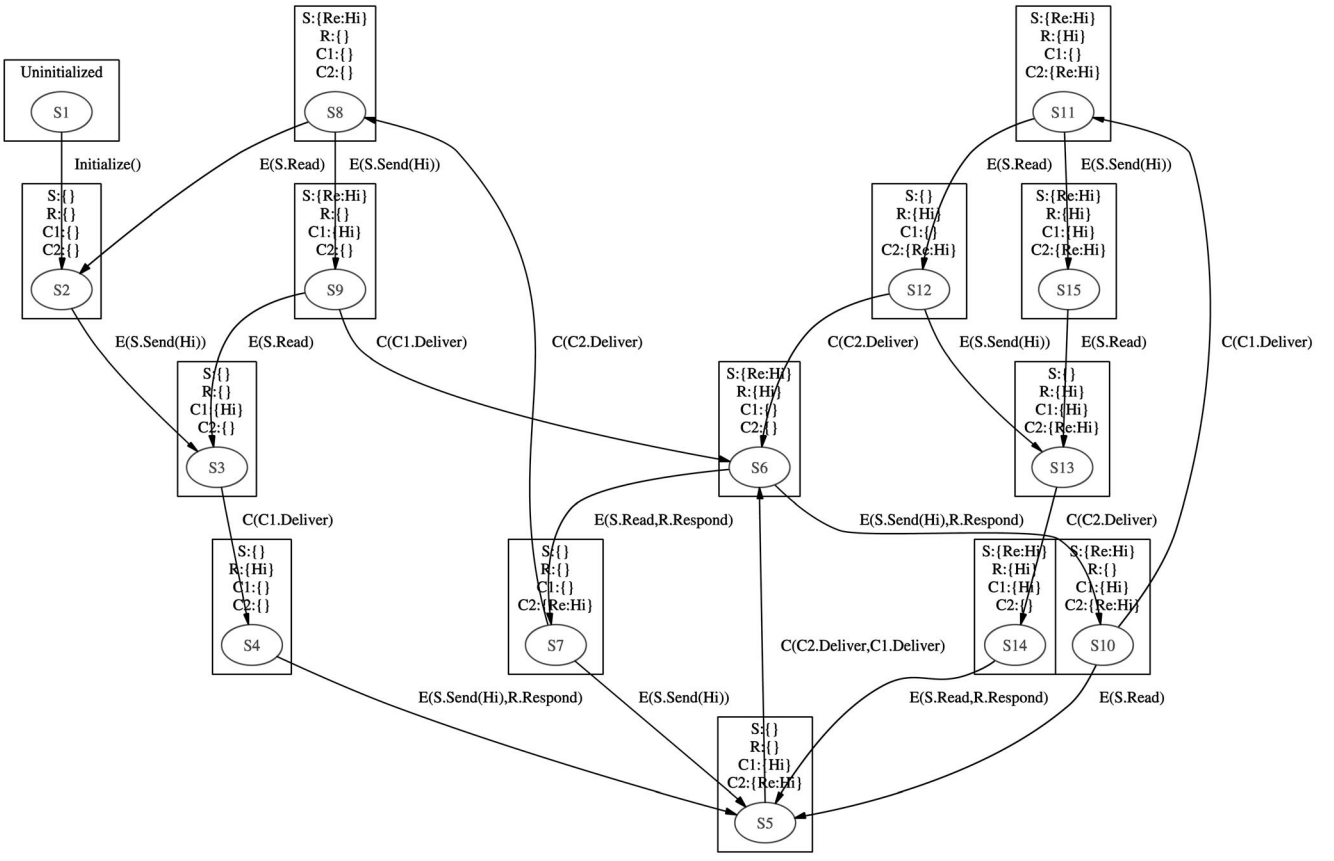


Fig. 5. FSM generated from the Sender-Responder model.

actions. A trace through the FSM corresponds to a run of the system that can be used as a scenario or a test case. In order to avoid irrelevant[6] interleavings of the agents in this model, the exploration is configured so that the enabled endpoint actions (one enabled action per endpoint agent) are fired simultaneously in a single transition; similarly for communicators. This is justified by the fact that endpoints in this model are mutually independent (they do not share state), the same holds for communicators. Notice that mutual independence does not hold for all pairs of the agents. For example, `C2` writes into sender's mailbox and sender reads from it; the resulting state depends on the order these actions are executed.

By running the model explorer on this model with the given settings, we get an FSM as shown in Fig. 5. Transitions that describe endpoint and communicator actions are labeled with `E(...)` and `C(...)`, respectively. For example, from state S6 the transition labeled `E(S.Read,R.Respond)` (both Read and Respond are fired) goes to state S7 and the transition labeled `E(S.Send(Hi),R.Respond)` goes to state S10. The specific message `Hi` originates from the parameter configuration settings of the tool. The values of the mailboxes in all the states except for the uninitialized one are shown in the boxes surrounding the nodes that represent the corresponding states (Fig. 5).

After generating the FSM, the tool uses standard FSM based techniques to produce test sequences. For example, in our case, we produce a 43 step long minimal transition tour (so-called Chinese postman traversal) of the whole FSM. See the beginning of the tour as shown in Fig. 6.

In real applications, such as Indigo [13], that are designed to operate reliably under less than ideal networking conditions, it is often desirable to explore the model behavior under corresponding conditions. A typical goal is test case generation. It has been our experience that often such network conditions can be realized in the model by appropriately adapting the behavior of the communicators, without changing the remainder of the model. For example, in our sample model we could allow the communicator `C2`, from the responder to the sender, to lose some of its messages.

7 Related Work

We address four bodies of work: Architecture Description Languages, Network Simulation Tools, Coordination Languages, and ASM models.

7.1 Architecture Description Languages

The main focus of an ADL is to specify a system's *conceptual architecture* rather than its *actual implementation*. Recent

6. What is or is not relevant depends on the purpose of the exploration. Interleavings that seem irrelevant may turn out to be relevant due to possibly unknown dependencies in the actual system that are not reflected in the model.

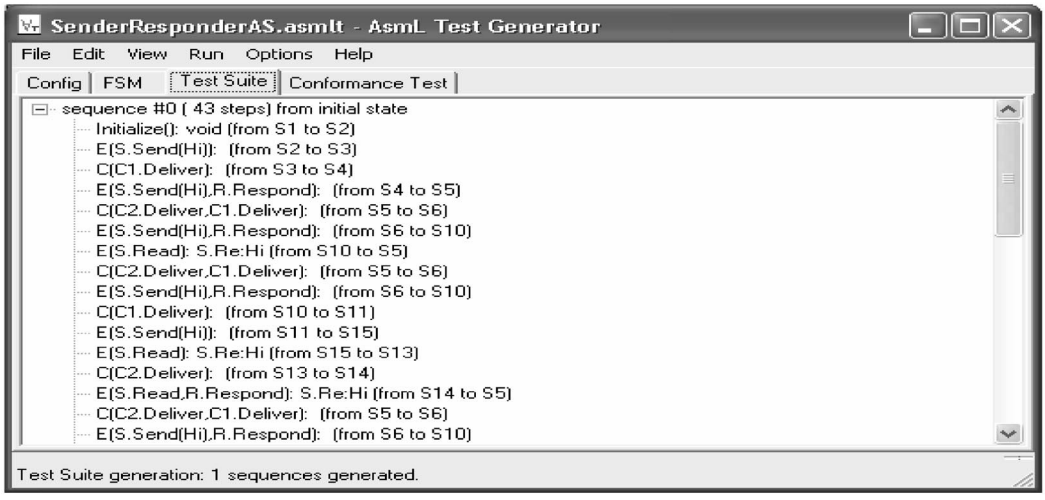


Fig. 6. Initial segment of test sequence generated from the Sender-Responder model.

surveys of ADLs are found in [14] and [35]. The following is a quote from [35] regarding the need for ADLs:

> They [ADLs] are necessary to bridge the gap between informal, "boxes and lines" diagrams and programming languages which are deemed too low-level for application design activities.

From that point of view, AsmL is an ADL, and the abstract communication model can be seen as an ADL artifact. There are, however, important distinctions between AsmL and the traditional ADLs. The most important distinction is that AsmL is executable.

AsmL has rigorous mathematical semantics. This is in contrast to many existing ADLs which lack formal semantics completely, or use different formal semantic models for components and connectors [35]. A rigorous semantics is often a prerequisite for many tools [32], [31] and for being able to reason about the global behavior of the modeled system. Since AsmL specifications are executable, they can be and are used for simulation, automatic test case generation [24], [8], scenario-based testing [7], conformance checking [6], [9], to provide behavioral interfaces for components [5], and so on. There are ongoing efforts in combining AsmL with automated or semiautomated verification tools. The issue of ASMs versus ADLs was addressed in [37], [38].

7.2 Network Simulation Tools

Among the simulation tools targeted at networking research, the most prominent and widely used one is the *ns*-2 network simulator [36] developed by the VINT project [41], a collaboration of UC Berkeley, LBL, USC/ISI, and Xerox Parc. Ns-2 is a powerful discrete event simulator written in a mixture of C++ and OTcl. The latter is an object oriented dialect of Tcl. Ns-2 provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless networks.

It may seem that ns-2 models and abstract communication models (ACMs) compete, but we do not think so. The two approaches target complementary aspects of computer network modeling. Basically, the difference is between a light-weight specification and a heavy-weight prototype; both are needed. A heavy-weight ns-2 simulates real-world systems as realistically as possible. The inherent complexity of ns-2 models makes it difficult for the user to focus on purely semantic aspects of the model. A light-weight, plug-in ACM-based simulator facilitates the analysis of the interaction between network entities and application entities as required for modeling and analyzing distributed systems at a semantic level.

As we see it, every distributed system, properly abstracted, gives rise to an ACM, a faithful abstraction that does not commit you to any implementation decisions. An ACM can be viewed as a high-level specification of the communication part of the global system. This piece of theory helps one comprehend and design a distributed system. An ACM is written in a simple high-level language and, thus, lends itself to analysis. On early design stages, it may be more appropriate for simulation. Note that "*users are responsible for verifying for themselves that their simulations are not invalidated because the model implemented in the simulator is not the model that they were expecting*" [36]. Such a verification is easier for an ACM-based simulator. Furthermore, an ACM model enables test case generation and global-level system analysis, e.g., via model checking. On the later stages, in particular, when timing issues are taken into account, one needs a heavy-duty tool like ns-2.

7.3 Coordination Languages

Coordination languages have been attracting much attention recently. The most famous coordination approach is that of Linda [19]. Here is a textbook on the subject: [1]. Some recent work (of group that interacts with our group) is found in [2]. The purpose of coordination languages is to give you means to program distributed systems. To do that, a coordination language provides a communication model. For example, in Linda, the coordination is achieved by means of operations on tuples that live in a (conceptually) shared space. Since the communication modes of a coordination language is used for programming, it's abstraction level is fixed. Our language, AsmL, is not

aimed for programming distributed systems and the abstraction level of our abstract communication model is much higher. As a result, the task of analysis is much easier in our case. Besides, a coordination language operates on top of a conventional programming language, like C, whereas AsmL is a single language.

7.4 ASM Models

The abstract communication model is not the first ASM model related to network communication. There exist DASM models of network protocols, including well-known protocols like Kermit [33] and Kerberos [10]. However, none of those models explicitly separates the behavior of the network from the behavior of the particular application as is done by our abstract network model. To this end, the novel idea presented here is to make the network model reusable.

8 Conclusion

We presented the abstract communication model (ACM). It is our contention that every message-based distributed system, properly abstracted, gives rise to a specialization of the ACM. The ACM is potentially useful for system level analysis of the given distributed system. It separates the interaction logic from the computation logic. It is not dissimilar to using a coordination language except that the appropriate specialization of the ACM abstracts the irrelevant details and is tailored to fit the semantics of the application in hand. There are some limitations of course. The ACM is useful mostly at the beginning of the modeling process; it allows one to conceptualize (and document!) the communication part of the system and, thus, helps you to see the forest behind the trees. The ACM abstracts from timing issues. The ACM time is logical; it is not obvious how to incorporate real time into the picture.

We have been mostly concerned with verifying our contention. As far as the analysis goes, we have been using the ACM only for simulation and test-generation. In these usages, the ACM itself is not very interesting or informative; it is the interplay between (a specialization of) the ACM and the particular application that is of interest. A relatively modest size of a typical ACM specialization makes it amenable for other types of analysis, e.g., model checking. We have not done that yet. As far as performance analysis is concerned, it is not clear whether the ACM is useful at all for the purpose.

One can imagine that the ACM may be useful for high-level design of distributed systems; we have done nothing in this direction.

Acknowledgments

The authors would like to thank the anonymous referees for valuable critique, comments, and suggestions on an earlier draft of this paper. The work on which this paper is based was done when U. Glässer visited Microsoft.

References

[1] J.-M. Andreoli, C. Hankin, and D. Le Métayer, *Coordination Programming: Mechanisms, Models and Semantics.* World Scientific, 1996.

[2] F. Arbab and F. Mavaddat, "Coordination Languages and Models," *Proc. Coordination 2002 Conf.,* Apr. 2002.

[3] Abstract State Machines (ASMs), http://www.eecs.umich.edu/gasm, 2003.

[4] AsmL for Microsoft.NET, http://www.research.microsoft.com/foundations/asml, 2003.

[5] M. Barnett and W. Schulte, "Runtime Verification of .NET Contracts," *J. Systems and Software,* vol. 65, no. 3, pp. 199-208, 2002.

[6] M. Barnett, C. Campbell, W. Schulte, and M. Veanes, "Specification, Simulation and Testing of COM Components Using Abstract State Machines," *Formal Methods and Tools for Computer Science, Proc. Eurocast 2001 Conf.,* pp. 266-270, Feb. 2001.

[7] M. Barnett, W. Grieskamp, Y. Gurevich, W. Schulte, N. Tillmann, and M. Veanes, "Scenario-Oriented Modeling in AsmL and Its Instrumentation for Testing," *Proc. Int'l Conf. Software Eng. (ICSE/SCESM),* 2003.

[8] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Towards a Tool Environment for Model-Based Testing with AsmL," *Proc. Formal Approaches to Software Testing Conf. (FATES 2003),* 2004.

[9] M. Barnett, L. Nachmanson, and W. Schulte, "Conformance Checking of Components Against Their Non-Deterministic Specifications," Technical Report MSR-TR-2001-56, Microsoft Research, June 2001.

[10] G. Bella and E. Riccobene, "Formal Analysis of the Kerberos Authentication System," *J. Universal Computer Science,* vol. 3, no. 12, pp. 1337-1381, 1997.

[11] A. Blass and Y. Gurevich, "Abstract State Machines Capture Parallel Algorithms," *ACM Trans. Computational Logic,* vol. 4, no. 4, pp. 578-651, Oct. 2003.

[12] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer, 2003.

[13] D. Box, "Code Name Indigo: A Guide to Developing and Running Connected Systems with Indigo," *MSDN Magazine,* http://msdn.microsoft.com/msdnmag/, 2003.

[14] P. Clements, "A Survey of Architecture Description Languages," *Proc. Eighth Int'l Workshop Software Specification and Design,* Mar. 1996.

[15] D.E. Comer, *Internetworking with TCP/IP, Principles, Protocols, and Architectures.* Prentice Hall, 2000.

[16] R. Farahbod, U. Glässer, and M. Vajihollahi, "Specification and Validation of the Business Process Execution Language for Web Services," Technical Report SFU-CMPT-TR-2003-06, Simon Fraser Univ., Sept. 2003.

[17] R. Farahbod, U. Glässer, and M. Vajihollahi, "Specification and Validation of the Business Process Execution Language for Web Services," *Proc. ASM Conf. 2004,* 2004.

[18] Foundations of Software Engineering Group at Microsoft, http://research.microsoft.com/fse, 2003.

[19] D. Gelernter and N. Carriero, "Coordination Languages and Their Significance," *Comm. ACM,* vol. 35, no. 2, pp. 97-107, Feb. 1992.

[20] U. Glässer, Y. Gurevich, and M. Veanes, "Universal Plug and Play Machine Models, Foundations of Software Engineering," Technical Report MSR-TR-2001-59, Microsoft Research, Redmond, June 2001.

[21] U. Glässer, Y. Gurevich, and M. Veanes, "High-Level Executable Specification of the Universal Plug and Play Architecture," *Proc. 35th Hawaii Int'l Conf. System Sciences (HICSS-35),* Jan. 2002.

[22] U. Glässer and M. Veanes, "Universal Plug and Play Machine Models: Modeling with Distributed Abstract State Machines," *Proc. IFIP World Computer Congress, Stream 7 on Distributed and Parallel Embedded Systems (DIPES'02),* Aug. 2002.

[23] U. Glässer, R. Gotzhein, and A. Prinz, "The Formal Semantics of SDL-2000: Status and Perspectives," *Computer Networks,* vol. 42, no. 3, pp. 343-358, June 2003.

[24] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating Finite State Machines from Abstract State Machines," *Proc. Int'l Symp. Software Testing and Analysis, Software Eng. Notes,* vol. 27, no. 4, pp. 112-122, 2002.

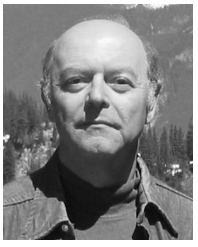
[25] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide," *Specification and Validation Methods,* E. Börger, ed., pp. 9-36, 1995.

[26] Y. Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms," *ACM Trans. Computational Logic,* vol. 1, no. 1, pp. 77-111, July 2000.

[27] Y. Gurevich and N. Tillmann, "Partial Updates: Exploration," *J. Universal Computer Science,* vol. 11, no. 7, pp. 917-951, 2001.

[28] Y. Gurevich and N. Tillmann, "Partial Updates Exploration II," to appear.
[29] Y. Gurevich, B. Rossman, and W. Schulte, "Semantic Essence of AsmL," Microsoft Research Technical Report MSR-TR-2004-27, Mar. 2004.
[30] Y. Gurevich and W. Schulte, "Semantics of AsmL," *Proc. Second Int'l Symp. Formal Methods for Components and Objects,* Nov. 2003.
[31] J. Heering, "Application Software, Domain-Specific Languages, and Language Design Assistants," *Proc. SSGRR 2000 Int'l Conf. Advances in Infrastructure for Electronic Business, Science, and Education on the Internet,* May 2000.
[32] J. Heering and P. Klint, "Semantics of Programming Languages: A Tool-Oriented Approach," *ACM SIGPLAN Notices,* vol. 35, no. 3, pp. 39-48, Mar. 2000.
[33] J.K. Huggins, "Kermit: Specification and Verification," *Specification and Validation Methods,* E. Börger, ed., Oxford Univ. Press, pp. 247-293, 1995.
[34] A. Prinz and M.v. Löwis, "Generating a Compiler for SDL from the Formal Language Definition," *Proc. Specification and Description Language (SDL) Forum,* 2003.
[35] N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Software Eng.,* vol. 26, no. 1, pp. 70-93, Jan. 2000.
[36] NS-2 Reference Manual, http://www.isi.edu/nsnam/ns/, 2003.
[37] A. Sünbül, "Abstract State Machines for the Composition of Architectural Styles," *Proc. Third Int'l Conf. Perspectives of System Informatics,* July 1999.
[38] A. Sünbül, "Architectural Design of Evolutionary Software Systems in Continuous Software Engineering," PhD thesis, Technical Univ. Berlin, Der Andere Verlag, Sept. 2001.
[39] "UPnP Device Architecture V1.0," *Microsoft Universal Plug and Play Summit,* Jan. 2000.
[40] Official Web site of the UPnP Forum, www.upnp.org, 2003.
[41] Virtual InterNetwork Testbed (VINT), A Collaboration among USC/ISI, Xerox PARC, LBNL, and UCB, http://www.isi.edu/nsnam/vint/index.html, 2003.
[42] Web Service Description Language (WSDL), W3C Note, Mar. 2001, www.w3.org/TR/wsdl.
[43] "XLANG Abstract Machine, Foundations of Software Engineering," internal report, Microsoft Research, 2002.
[44] "XLANG: Web Services for Business Process Design," www.gotdotnet.com/team/xml_wsspecs/xlang-c, 2003.

Uwe Glässer received the MSc, PhD, and Habil. (postdoctoral qualification) degrees in computer science, respectively, from the University of Paderborn, Germany. He is an associate professor of computing science and the founder of the Software Engineering Lab at Simon Fraser University. His research interests focus on foundations of software engineering and their applications in industrial systems design.



Yuri Gurevich is a senior researcher at Microsoft Research in Redmond, WA. He is also Professor Emeritus at the University of Michigan, an ACM fellow, Guggenheim fellow, and Dr. Honoris Causa of Limburg University in Belgium.

Margus Veanes received the masters degree in computer science from Uppsala University in 1990, where he also defended his PhD in computer science in 1997. Since 1999, he has been a researcher in the Foundations of Software Engineering Group at Microsoft Research. His main research interests are in model-based software development and testing.