An Abstract Communication Model

Uwe Glässer[1], Yuri Gurevich and Margus Veanes

May 2002

Technical Report
MSR-TR-2002-55

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Abstract

We present an abstract communication model. The model is quite general even though it was developed in the process of specifying a particular network architecture, namely the Universal Plug and Play (UPnP) architecture. The generality of the model has been confirmed by its reuse for different architectures. The model is based on distributed abstract state machines and implemented in the specification language AsmL.

Keywords: Distributed Systems, Computer Networks, Communications Software, Requirements Specification, Systems Modeling, Rapid Prototyping

---

[1] This author is currently at Heinz Nixdorf Institute, Paderborn, Germany. The work on which this paper is based was done mainly when he visited Microsoft.

1 Introduction

The group on Foundations of Software Engineering at Microsoft Research [16] has developed a high-level executable specification language AsmL [2] based on the concept of *abstract state machine* or ASM [21]. AsmL is integrated with Microsoft's software development, documentation and runtime environments. AsmL supports specification and rapid prototyping of object oriented and component oriented software. It is a successful practical instrument for systems design (and reverse engineering).

ASMs are able to simulate arbitrary algorithms in the step-for-step manner. There is a substantial experimental confirmation [1][16] as well as theoretical confirmation [6][22] of that ASM thesis. ASMs have been used to specify various architectures, protocols and numerous languages, in particular C [20], Java [30], SDL [15] and VHDL [8]. The International Telecommunication Union adopted a comprehensive ASM-based formal definition of SDL as an integral part of the current SDL standard [28]. AsmL specifications look like pseudo code over abstract data structures. As such, they are easy to read and understand by system engineers and program developers. Practical experiences with industrial applications helped to establish a pragmatic understanding of how to model complex system behavior with a degree of detail and precision as needed [7][8].

Some features of high-level rigorous specifications are well recognized in the academic community as advantageous. 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 at [2]. AsmL is used to explore and test the design, to validate the specification itself, and to test the conformance of the implementation to the specification. In particular, executable specifications are used for test-case generation [25] and runtime verification [3].

And there are features of high-level rigorous specifications, at least of ASM specifications, that have not been given sufficient attention. 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 [17]. While the informal documentation 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 general and was reused on several occasions. The communication model partitions the whole system into a collection of communicating subsystems. The particular subsystems change from one project to another but the communication structure is the same. In particular, the communication model was reused in our work on XLANG [36] where the documentation given to us was partially formal but did not address the communication model.

In this paper, we present our communication model and illustrate it on the examples of UPnP and XLANG. As far as *domain specific languages* [14] are concerned, our work fits in well, the specific domain being Software Architecture Description.

We try to make this paper self-contained. In Section 2, we recall some basic definitions on abstract state machines (ASMs) and in particular distributed abstract state machines (DASMs). In Section 3, we describe a portion of AsmL sufficient for our purposes in this paper. The abstract communication model is described in Section 4. The abstract communication model is reused in both Section 5, where we describe the UPnP model, and in Section 6, where we describe the XLANG model. In Section 7 we discuss the classification of AsmL as a software architecture description language.

2 Abstract State Machines

Our method rests on the ASM theory. Here we give a quick description of ASMs sufficient for our purposes in this paper. The interested reader may want to consult [6][22][23].

2.1 Basic ASMs

A *basic* ASM consists of a basic ASM program together with a collection of states (the legal states of the ASM) and subcollection of initial states. First we describe basic ASM states and then define basic ASM programs.

Basic ASMs are sequential algorithms. Intuitively sequential algorithms are non-distributed 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 [23] where it is proved that, for every sequential algorithm, there is a basic ASM that simulates the algorithm step for step.

2.1.1 States

The notion of ASM state is a variation of the notion of (first-order) structure in mathematical logic.

A *vocabulary* is a collection of *function symbols* and *relation symbols* (or *predicates*); each symbol has a fixed arity (the number of arguments). Symbols split into *dynamic* and *static*. Every vocabulary contains (static) logic symbols *true*, *false*, *undef*, the equality symbol, and the standard propositional connectives.

A *state A* of a given vocabulary is a nonempty set *X* (the *base set* of *A*), together with interpretations of the function symbols (the *basic functions* of *A*) and the predicates (the *basic relations* of *A*). A function (respectively relation) symbol of arity *j* is interpreted as a *j*-ary operation (respectively relation) over *X*. A nullary function symbol is interpreted as an element of *X*. The logic symbols are interpreted in the obvious way.

The value *undef* is the default value for basic functions. Formally a basic function *f* is total but intuitively it is partial. The intended domain of *f* consists of all tuples $\boldsymbol{a}$ with $f(\boldsymbol{a}) \neq undef$. Every state includes an infinite "naked" set called the *reserve*. The role of reserve will become apparent later.

The default value for relations is *false*. We think of a *j*-ary relation *R* as a set of *j*-tuples of elements.

Remark. Traditionally, in logic, *true* and *false* do not belong to the base set, and so there is a principal difference between basic functions (taking values inside the structure) and basic relations (taking values outside). In our framework, basic relations are seen as special basic functions whose only possible values are *true* and *false* and whose default value is *false* rather than *undef*.

This simple definition of state is very general. Any kind of static mathematical reality can be described as a first-order structure. Second-order

and higher-order structures of logic are special first-order structures. Many-sorted first-order structures (with several base sets called sorts) are special one-sorted structures; the roles of sorts are played by designated unary relations which are called *universes* in the ASM literature. Notice that ASM universes may be dynamic.

Example 1. The vocabulary consists of one static unary predicate *Scandinavia* and one dynamic binary predicate *Flight*. (In addition, it contains the logic symbols but it is customary to ignore the logic symbols, their interpretation and the reserve in the descriptions of states.) The base set of our state consists of three airports ARN, CPH and SEA. The unary relation *Scandinavia* contains ARN and CPH but not SEA. The relation *Flight* is this

{(ARN,CPH),(CPH,ARN),(CPH,SEA),(SEA,CPH)}

The intended meaning is that there are direct flights (of some fixed airline) from ARN to CPH, from CPH to ARN, etc., but there are no direct flights between ARN and SEA.

2.1.2 Updates

We view a state as a kind of memory. Dynamic functions are those that can change during computation. A *location* of a state A is a pair $l = (f, (x_1,\ldots, x_j))$ where f is a j-ary dynamic function (or relation) symbol in the vocabulary of A and $(x_1,\ldots, x_j)$ is a j-tuple of elements of A. The element $y = f(x_1,\ldots,x_j)$ is the *content* of that location.

An update of state A is a pair (l, y'), where l is a location $(f, (x_1,\ldots, x_j))$ of A and y' is an element of A; of course y' is *true* or *false* if f is a predicate. To fire the update (l, y'), replace the old value $y = f(x_1,\ldots, x_j)$ at location l with the new value y' so that $f(x_1,\ldots, x_j) = y'$ in the new state. Intuitively, one may view dynamic functions as being represented by function tables that can be updated dynamically at run time. The effect of the update instruction above is illustrated in Figure 1; the new content y' is replaced by the old content y of location l.

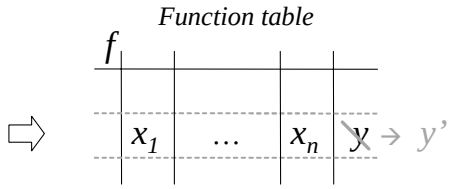

Figure 1. Function update

A set $S = \{(l_1, y'_1), ..., (l_n, y'_n)\}$ of updates is *consistent* if the locations are distinct. In other words, S is *inconsistent* if there are i, j such that $l_i = l_j$ but y'_i is distinct from y'_j. To fire a consistent set of updates, fire all the updates at once; to fire an inconsistent update set, do nothing.

2.1.3 Rules and Programs

Expressions are defined inductively. If f is a j-ary function symbol and $e_1,..., e_j$ are expressions then $f(e_1,..., e_j)$ is an expression. (The base of induction is obtained when $j = 0$.) If f is a predicate then the expression is *Boolean*.

An *update rule R* has the form

$f(e_1,...,e_j) := e_0$

where f is a j-ary dynamic function symbol and each e_i is an expression. (If f is a predicate then e_0 should be a Boolean expression.) To execute R, fire the update (l, a_0) where $l = (f, (a_1,..., a_j))$ and each a_i is the value of e_i.

A *conditional rule R* has the form:

if e **then** R_1 **else** R_2

where e is a Boolean expression and R_1, R_2 are rules. To execute R, evaluate the guard e. If e is true, then execute R_1; otherwise execute R_2.

A *do-in-parallel rule R* has the form

do in-parallel
R_1
R_2

where R_1, R_2 are rules. To execute R, execute rules R_1, R_2 simultaneously.

An *import rule R* has the form

import x
$R_1(x)$

To execute R, fish out any element x of the reserve and execute the rule $R_1(x)$.

A program (that is a basic ASM program) is just a rule. An ASM is given by a program, a collection of legal states and a subcollection of initial states. Note that the program described just one step of the ASM. It is supposed to be executed until – if ever – the state does not change.

Example 1 (continuation). The following rule reflects that direct flights have been established between ARN and SEA.

do in-parallel
Flight(ARN,SEA) := *true*
Flight(SEA,ARN) := *true*

2.2 Parallel ASMs

We generalize the definition of basic ASMs in a two directions.

First, we enrich the notion of expression.

$\{t(x) \mid x \in s$ **where** $\varphi(x)\}$

is an expression (a *comprehension expression*) denoting the set of all values $t(x)$ where x ranges over those elements of set s that satisfy $\varphi(x)$. This presumes that s is a set expression and $\varphi(x)$ is Boolean. We require that every state A is closed under tuples and (finite) sets: if $a_1,..., a_n$ are elements of A then the tuple $(a_1,..., a_n)$ and the set $\{a_1,..., a_n\}$ are elements of A. We require also that A contains the standard operations over tuples and sets, e.g. the set pairing operation $\{e_1, e_2\}$.

Remark. We often require also that the states are closed under finite partial maps but the set background is sufficient for theoretical purposes [6].

Second, we enrich the notion of rules. A *do-forall rule R* has the form

forall $x \in s$
$R_1(x)$

where $R_1(x)$ is a rule and x does not occur freely in the expression r. To execute R, execute all subrules $R(x)$ with x in s at once.

Parallel ASMs are parallel algorithms. The appropriate notion of parallel algorithms is formalized in [6] where it is proved that, for every parallel algorithm, there is a parallel ASM that simulates the given algorithm step for step.

2.3 Nondeterministic ASMs

Basic and parallel ASMs can be made nondeterministic by the use of *external basic functions*. For example, an ASM can employ a function *Input* operated

by the user. It is convenient though to have explicit nondeterminism. To this end, we enrich parallel ASMs with choose rules.

A *choose rule R* has the form

choose $x \in s$
  $R_1(x)$

where $R_1(x)$ is a rule and x does not occur freely in the set expression s. To execute R, choose any element x of s and execute the subrule $R_1(x)$.

Example 1 (continuation). Imagine that, for some reason, you want to remove all direct flights between SEA and one of the Scandinavian airports. The following rule accomplishes that.

choose $x \in$ {ARN,CPH}
  Flight(SEA, x) := *false*
  Flight(x, SEA) := *false*

2.4 Distributed ASMs

Until now, we dealt with one-agent ASMs. A *distributed ASM* (DASM) involves a collection of *agents*. Mathematical abstraction allows us to speak about *global states* of a DASM even though different agents may live on different computers. In a global state, the agents interact by reading from and writing into "shared" locations of the global state; potential read-write and write-write conflicts are resolved according to the definition of partially ordered runs below.

Agents are represented in global states as well. They are elements of a dynamically universe *Agent* that may grow and shrink. With each agent we associate a *program* defining its behavior. A static universe *Program* abstractly represents the set of all agent programs collectively forming the distributed program of A. Agents may perform their computation steps concurrently. A single computation step of an individual agent is called a *move* of this agent.

Formally, a *run* ρ of a distributed ASM A is given by a triple (M, λ, σ) satisfying the following four conditions:

1. M is a partially ordered set of moves where each move has only finitely many predecessors.
2. λ is a function on M associating agents with moves such that the moves of any single agent of A are linearly ordered.

3. σ assigns a state of A to each initial segment Y of M, where $\sigma(Y)$ is the result of performing all moves in Y; $\sigma(Y)$ is an initial state if Y is empty.

4. The *coherence condition*: If x is a maximal element in a finite initial segment X of M and $Y = X - \{x\}$ then $\lambda(x)$ is an agent in $\sigma(Y)$ and $\sigma(X)$ is obtained from $\sigma(Y)$ by firing $\lambda(x)$ at $\sigma(Y)$.

Intuitively, a run can be seen as the common part of histories of the same computation recorded by various observers. See [22] for further details.

Example 2. We illustrate the coherence condition in a simple situation. Suppose that we have only two propositional variables (dynamic nullary relation symbols) *door* and *window*. Intuitively *door = true* means that "the door is open" and *window = true* means that "the window is open". Imagine now two distinct agents: a *door-manager (*agent *d)* and a *window-manager (*agent *w)*. The program of the door-manager is to open the door if the window is closed (move *x*). The program of the window-manager is to open the window if the door is closed (move *y*).

$Program_d$ = **if** $\neg$*window* **then** *door* := true

$Program_w$ = **if** $\neg$*door* **then** *window* := true

Assume that initially (in state S_0) both the door and the window are closed. Then there are only two possible runs, and in each run only one of the agents makes a move. Indeed, we cannot have $x < y$ because w is disabled in the state S_1 obtained from S_0 by performing x. Similarly we cannot have $y < x$ because d is disabled in the state S_2 obtained from S_0 by performing y. Finally, we cannot have a run where x and y are incomparable, that is neither $x < y$ nor $y < x$. By the coherence condition, the final state S_3 of such a run would be obtained from S_1 by performing y which is impossible. (It would also be obtained from S_2 by performing x which is equally impossible.)

3 From Abstract State Machines to AsmL

To actually program ASMs in 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 [2].

First we explain how the fundamental modeling concepts of ASMs are realized in AsmL. 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 or simulation of true concurrency.

Remark. There is a fair amount of freedom in AsmL regarding the representation of ASM functions and domains. The reader should keep in mind that the particular choice of representation is often a matter of taste, readability and a way to control how the model is executed, and does not affect the underlying ASM semantics.

3.1 Types

Some ASM universes give rise to *types* in AsmL. Other universes are represented as (finite) sets; some examples are found below. An AsmL model may first declare an abstract type *C* and later on concretize that type into a *class*, a *structure*, a finite *enumeration*, or a derived type.

```
type C
class C
```

AsmL has an expressive type system that allows one to define new types using (finite) sets, (finite partial) maps, (finite) sequences, tuples, etc., combined in arbitrary ways. For example, if *C* and *D* are types then the type of all maps from *C* to sets of elements of *D* is this.

```
Map of C to Set of D
```

Finite sets, sequences, maps are ordinary elements. The common operations on sets, sequences, maps and other built-in data types are available as built-ins, for example the binary operation *apply*(*f*, *a*) applies a map *f* to an element *a*. The shorthand notation for map application is *f*(*a*).

3.2 Derived Functions

Derived functions play an important role in applications of ASMs. A derived function does not appear in the vocabulary; instead it is computed on the fly at a given state. In AsmL, derived functions are given by methods that return values. A derived function *f* from $C_0 \times C_1 \times \ldots \times C_n$ to *D* can be declared as a global method.

```
f(x0 as C0, x1 as C1, ..., xn as Cn) as D
```

The definition of (how to compute) *f* may be given together with the declaration or introduced later on in the code. Alternatively, if C_0 is a class (or a structure) then *f* can be declared as a method of C_0. Notice that *n* can be *0*.

```
class C0
  f(x1 as C1,...,xn as Cn) as D
```

A nullary derived function can be introduced as a global method that takes no arguments. For example

```
z() as Integer return e
```

where *e* is evaluated in a given state.

3.3 Constants

A nullary function that does not change during the evolution can be declared as a *constant*.

```
z as Integer = 0
```

A unary static function from a class *C* to *D* can be declared as a constant field of *C* as in following example.

```
class C
  id as String
```

3.4 Variables

There are two kinds of variables, *global variables* and *local variable fields* of classes. Semantically, fields of classes are *unary* functions.

```
var b as Boolean
class C
  var f as Integer
```

Notice that *f* represents a unary dynamic function from *C* to integers.

Dynamic functions of ASMs are represented by variables in AsmL. A dynamic function *f* from $C_1 \times ... \times C_n$ to *D* of any positive arity *n* can be represented as a variable map in AsmL.

```
var f as Map of (C1, ..., Cn) to D
```

With the map representation, a normal ASM update $f(\boldsymbol{c}) := b$ corresponds to a partial update of the map variable *f*. A set of consistent ASM updates to *f* corresponds to a set of consistent (non-conflicting) partial map updates that are combined into a single total update of *f*. We do not use partial updates on maps in this paper. The theory of partial updates is developed in [23].

3.5 Classes and Dynamic Universes

AsmL classes are special *dynamic universes*. Classes are initially empty. Let *C* and *D* be two dynamic universes such that *C* is a subset of *D* and let *f* be a dynamic function from *C* to integers.

```
class D
class C extends D
  var f as Integer
```

The following AsmL statement adds a new reserve element *c* to *C* and *D* and initializes *f*(*c*) to the value 0.

```
let c = new C(0)
```

Classes are special dynamic universes in that one cannot programmatically remove an element from a class. In general, classes cannot be quantified over like sets (given by expressions) but it is possible to check whether a given element is of type *C* by using the *is* keyword.

```
if x is C then ...
```

In order to keep track of elements of a class *C*, one can introduce (essentially an auxiliary dynamic universe represented by) a variable of type *set of C* that is initially empty.

```
var Cs as Set of C = {}
```

Set-valued variables can be updated partially by inserting and removing individual set members. Several such pairwise *non-conflicting partial updates* (i.e. you don't both insert and remove the same element) are combined into a single *total update* at the end of the step.

```
let c = new C(0)
```

```
Cs(c) := true    //insert c into Cs
```

Sets (given by expressions) can be quantified over like in the following rule where all the invocations of *R*(*x*), one for every element *x* of the set *s*, happen simultaneously in one atomic step.

```
forall x in s
  R(x)
```

3.6 Simulation of Agents

Ideally one would like to have a distributed runtime environment for running distributed ASMs. The current version of AsmL doesn't have yet runtime support for true concurrency or simulation of true concurrency. Therefore we have to build functionality for simulating concurrent agents into our model. This is how we do it.

Agents are introduced through a class *Agent*. To keep track of the currently active agents, a variable *Agents* of type *set of Agent* is updated each time an agent is created or discarded.

```
class Agent
var Agents as Set of Agent = {}
```

Each agent (more exactly, its state) evolves in sequential steps with each invocation of its *Program*. Each agent *a* has a *mailbox* of *messages* and a method *InsertMessage* that is used by other agents to send messages to *a*.

```
type Message
class Agent
  Program()
  var mailbox as Set of Message = {}
  InsertMessage(m as Message)
    mailbox(m) := true
```

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

The method *RunAgents* gives the operational semantics of a single step of the top-level system, in the definition of which *chooseSubset* selects nondeterministically a subset of the active agents. 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(Agents)
    Program(a)
```

4 Abstract Communication Model

It is not clear how to deal with the network in a sufficiently abstract and general way. 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 represents a part of the communication network. Intuitively different communicators represent disjoint subnets, and typically this is indeed the case. But we don't impose this restriction.

```
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
```

Our communication model is a distributed ASM. It was obtained by an abstraction of TCP/IP networks [13]. The multitude of communicators reflects the fact that a TCP/IP network consists of distinct interconnected physical networks. We abstract from routing details. The communicators transfer messages between applications. Figure 2 is a sketch of an instance of the abstract communication model. 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.

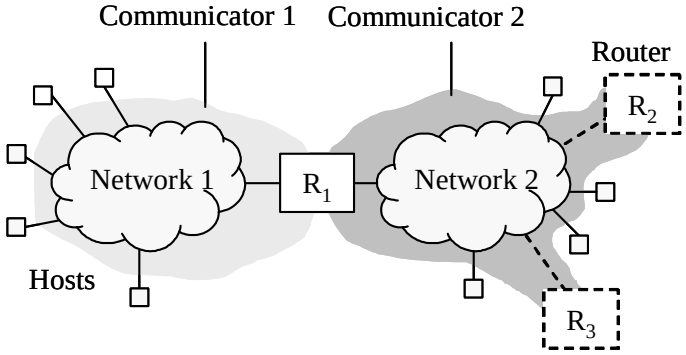

Figure 2. Communicators.

The whole system, applications and communicators, operates in some external environment which may affect the system in various ways. For example, it may change the system configuration by creating and removing agents. It may also affect the communication load of the network which affects message delays. Those external environmental actions and events are unpredictable as far as the system is concerned.

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 are 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 non-reliable behavior of the network.

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

4.1 Message Transformation

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

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 separate 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.

```
type MESSAGE
class COMMUNICATOR
   ResolveMessage(m as MESSAGE) as Set of MESSAGE
```

In the TCP/IP world, addressing mechanisms classify as *unicasting*, *broadcasting* or *multicasting* where multicasting can be viewed as the most general one [13]. 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 *addressTable* 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.

```
destination(m as MESSAGE) as ADDRESS
class COMMUNICATOR
  Transform(m as MESSAGE, dest as ADDRESS) as MESSAGE
  ResolveMessage(m as MESSAGE) as Set of MESSAGE
    return {Transform(m,a) | a in addressTable(destination(m))}
```

4.2 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 would receive an outbound message if that message were to be forwarded by the communicator. The recipient may be an application running on a local host 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 *undef*; this possibility forces us to use the type *Agent*? which consists of agents and the *undef* value.

```
class COMMUNICATOR
  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 (where the degree of detail and precision can be freely chosen depending on the given application context), is through a (possibly dynamic) mapping called *routingTable*. The routing table of a communicator maps 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. If there is no entry in the routing table for a given address *a* then the value of *routingTable*(*a*) is *undef*.

```
class COMMUNICATOR
  Recipient(m as MESSAGE) as Agent?
    return routingTable(destination(m))
```

4.3 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 message is ready to deliver in a given state of the network is expressed through an *external* predicate *ReadyToDeliver*.

```
class COMMUNICATOR
  external ReadyToDeliver(m as MESSAGE) as Boolean
```

Note that messages that are never ready to deliver are in effect "lost", even though they persist in the communicator's mailbox. (For example, for some UDP message *m* the condition *ReadyToDeliver*(*m*) might never hold.)

4.4 Message Delivery

We are now ready to present an algorithm for how communicators route messages to other agents. The control program of a communicator forwards messages found in its mailbox by inserting them into the mailboxes of the respective recipients of the message. The communicator program is highly nondeterministic.

```
class COMMUNICATOR
  Program() =
    let availableMsgs = {m | m in me.mailbox where ReadyToDeliver(m)}
    let selectedMsgs = chooseSubset(availableMsgs)

    forall msg in selectedMsgs
      me.mailbox(msg) := false               //delete the message
      let resolvedMsgs = ResolveMessage(msg) //resolve the message
      forall m in resolvedMsgs
        let a = Recipient(m)
        if a <> undef then                   // if recipient found
          InsertMessage(a,m)                 // forward the message
        else
          skip                               // else ignore message
```

First, the communicator determines the subset of unprocessed messages that are ready to be delivered. Next, the communicator (nondeterministically) selects a subset of the available messages for processing in this step. Note that some, all or none of the available messages 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.

5 Universal Plug and Play

The *Universal Plug and Play Architecture* (UPnP) [33] is an industrial standard for dynamic peer-to-peer networking defined by the UPnP Forum [34]. Here is how UPnP is described in [33]:

> *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 behavior model of the UPnP architecture [17][18][19] based on the informal requirements specification [33]. The construction of a DASM allows us to combine both *synchronous* (that is one-agent) execution models of individual devices and control points and *asynchronous* execution models of an ensemble of devices and control points within one uniform model of computation. Here we give an overview of our UPnP model focusing on some interoperability aspects (rather than on the internal behavior of UPnP components) related to the abstract communication model.

5.1 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: DHCP, SSDP, SOAP, GENA, etc. 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.

A sample UPnP device, a *CD player*, is shown in Figure 3. In the full model [17], this device has two different services, called *ChangeDisc*, and *PlayCD*; Figure 3 illustrates only the first one. The *ChangeDisc* service allows a control point to add or remove discs from the CD player, to choose a disc to

be placed on the tray, and to toggle (open/close) the door. The figure illustrates the relevant state information associated with the service.

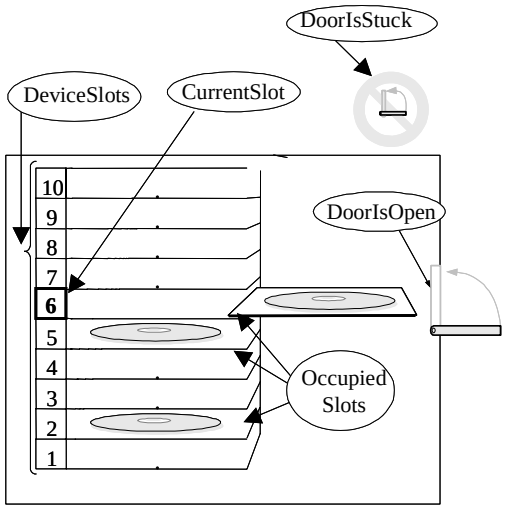

Figure 3. ChangeDisc service of a CD player.

5.1.1 Protocol Phases

The UPnP protocol defines 6 basic steps or phases. Initially, these steps are invoked one after the other in the order given below, but may arbitrarily overlap afterwards. (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 run time. (5) *Presentation* enables users to retrieve additional device vendor specific information.

5.1.2 Restrictions

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.

5.2 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 [17] employs some additional agents that reflect the external world, e.g. DHCP server agents, but here we ignore them.

With each agent type we 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) allowing for user-controlled interaction with the environment. The overall organization of the model is illustrated in Figure 4.

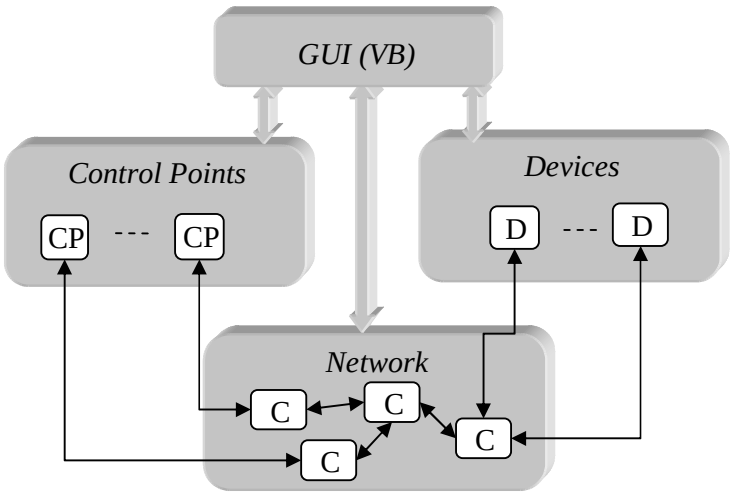

Figure 4. Instance of UPnP Abstract Machine.

Device model

The purpose of the device model specification is to describe how a device behaves in a 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 in an unpredictable way. The device model specification is a parallel composition of a number of rules operating in parallel; different rules describe different protocol phases.

```
external isConnected(d as DEVICE) as Boolean
class DEVICE
  Program()
    if isConnected(me) then
      RunAddressing()
      RunDiscovery()
      RunDescription()
      RunControl()
      RunEventing()
      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. Each service of a device can be called by 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 succeded or not and may include a 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. A device may receive messages from that communicator but also from other communicators. 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
  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 *undef* 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.

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

  var address as ADDRESS? = undef
  RunControl()
    if address ≠ undef then
      RunServices()
      choose msg in mailbox, s in services
             where IsServiceRequest(msg,s)
        reply = Call(s,msg)
        mailbox(msg) := false
        InsertMessage(communicator, reply)
```

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.

6 Modeling Automated Business Processes

In this section we summarize a real-life application of distributed abstract state machines to model automated business processes [36]. The purpose of the summary is to illustrate the effective reuse of the abstract communication model.

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.

6.1 XLANG

XLANG is an XML based formal language that can be used to define the data and networking protocols of automated business processes [32]. 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 [35]. 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.

6.2 XLANG Abstract Machine

The definition of an abstract operational semantics for XLANG comes in the form of an abstract machine model in combination with an XLANG-to-AsmL compiler. The behavior is formalized by mapping a given XLANG contract to AsmL code effectively explaining this behavior 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.

Each of the service behaviors is compiled into a sequence of *XLANG Abstract Machine* (XAM) instructions. The port map determines the routing information that is used by the network abstract machine to interconnect the services. The approach taken here is similar to the concept of "phases of compilation" in compiler construction.

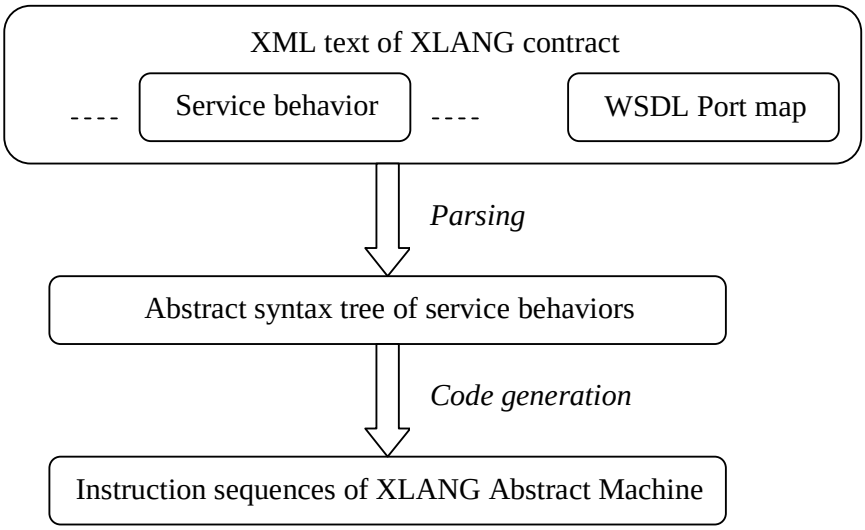

Figure 5. Generation of XAM instructions

Note that our approach is *abstract*: the structure we describe does reflect a real-world implementation but omits implementation-specific detail.

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.

6.2.1 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 sub-process spawned by a previously created process.

During its lifetime, a service instance may spawn several sub-processes. 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 (a) a process has exited by having run all of the XAM instructions, (b) a process has been interrupted by an exception, (c) a process has been halted by external intervention, or (d) a process is currently executing instructions.

```
type Service
type Label
type ServiceProgram
class Manager extends Agent
  service as Service
  pgm as ServiceProgram
enum ProcessMode
  exited
  raised
  halted
  running
```

```
class Process extends Agent
  manager as Manager
  var pc as Label
  var mode as ProcessMode
  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
Execute(intr as Instruction, p as Process)
instr(pgm as ServiceProgram, lbl as Label) as Instruction
class Process
  Program()
    if mode = running then
      let instr = instr(manager.pgm, pc)
      Execute(instr, me)
    else
      skip
```

A manager has two independent jobs. One is to activate new service instances when *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 is to handle message traffic

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

6.2.2 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 *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.

```
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.

```
class Communicator
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 ports where p.outbox ne {}
      choose m in p.outbox
        p.outbox := p.outbox - {m}
```

```
        let msg = new NetworkMessage(portMap(p),m)
        InsertMessage(p.communicator,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()
    if mailbox ne {} then
      choose m in mailbox
        mailbox := mailbox - {m}
        let p = m.port
        p.inbox := p.inbox union {m.msg}
```

Figure 6 shows an instance of the XLANG abstract machine.

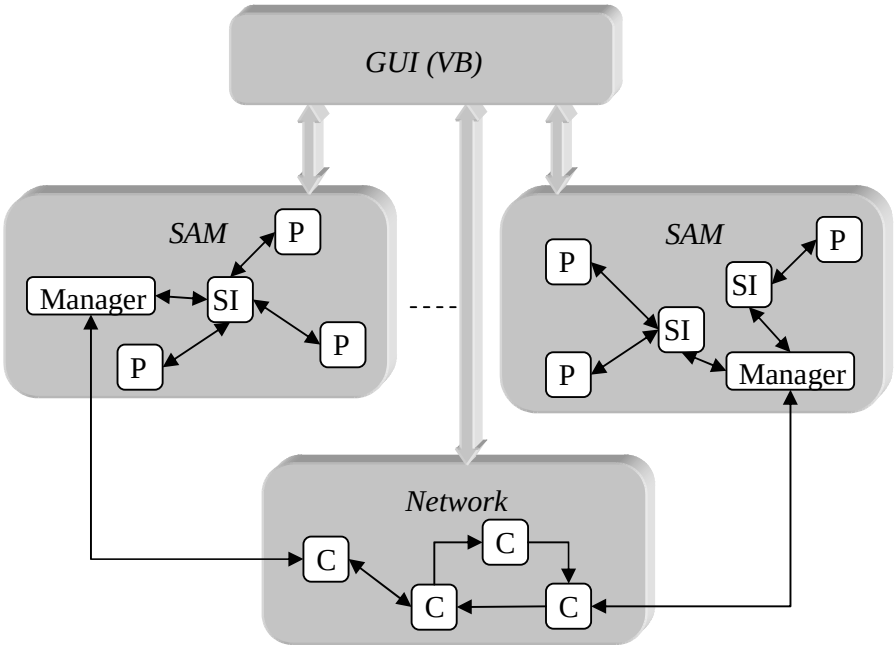

Figure 6. Instance of XLANG Abstract Machine

The XLANG model instance in Figure 6 contains several service abstract machines (SAMs) and a network abstract machine (NAM). Each SAM contains a manager, some service instances and other processes. The NAM contains several communicators. The XLANG abstract machine has been implemented in AsmL [36]; a GUI is used to interact with the model and visualize the state during simulation runs.

7 Related Work

We start with domain-specific languages; the connection to AsmL will soon become apparent. General introductions to domain-specific languages are given in [14][27]. The annotated bibliography [14] categorizes the domains of various domain-specific languages into five different groups. The group on *software engineering* is further subdivided into several subgroups including one for *software architectures*. The main focus of a software *architecture description language* (ADL) is to specify system's conceptual architecture rather than its actual implementation. Recent surveys of ADLs are given in [11] and [30]. This is a quote from [30] regarding the prevailing argument for using ADLs:

> *They 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.*

According to [29], an ADL must provide means for *explicit* specification of the following building blocks of an architectural description: *components*, *connectors*, and *configurations*. Let's see what these building blocks are in AsmL.

Components are agents or groups of agents together with a collection of interfaces defining the interaction points of the component with the environment. The interfaces may be declared as native COM [10] interfaces, automation interfaces or abstract model interfaces, depending on their usage. For example, in the UPnP model, devices are components that interact with communicators via abstract model interfaces and with the GUI via automation interfaces.

Connectors are special components that enable the interaction of other components. Their behavior is clearly separated from the core behavior of the model. For example, in the UPnP model the communicators are the connectors; indeed they do not reflect any UPnP specific behavior.

Configurations describe the topology of the system. In AsmL, configurations are normally described explicitly in the state. For example, the address table and the routing table in the abstract communication model encode configurations. However AsmL does not have an explicit configuration sublanguage considered necessary in [30].

The main strength of AsmL is the unified semantic model based on ASMs. This is in contrast to many existing ADLs which lack formal semantics completely, or use different formal semantic models for components and connectors [30]. A rigorous semantics is often a prerequisite for many tool generators [28]. AsmL specifications can be used for automatic test case generation [25], conformance checking [4][5], and to provide behavioral interfaces for components [3].

Methodological guidelines and epistemological reasons *how* and *why* the ASM paradigm offers a mathematically well founded approach to high-level systems design and analysis of complex system behavior, also in relation to other formal methods, are discussed in [7][8].

References

[1] Abstract State Machines (ASMs), the academic Web site, http://www.eecs.umich.edu/gasm.

[2] AsmL, the ASM Language, the website, htt:p//research.microsoft.com/fse/asml

[3] M. Barnett and W. Schulte. The ABCs of Specification: AsmL, Behavior, and Components, *Informatica*, 25(4), 2001.

[4] M. Barnett, C. Campbell, W. Schulte, and M. Veanes. Specification, simulation and testing of COM components using Abstract State Machines. In Formal Methods and Tools for Computer Science, Eurocast 2001, pp. 266-270. IUCTC Universidad de Las Palmas de Gran Canaria, February 2001.

[5] 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.

[6] A. Blass and Y. Gurevich. Abstract State Machines Capture Parallel Algorithms. Microsoft Research, Technical Report, MSR-TR-2001-117. To appear in *ACM Transactions on Computational Logic*, 2002.

[7] E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, M. Ullman, eds., Current

Trends in Applied Formal Methods (FM-Trends 98). Springer LNCS 1641, pp. 1-43, 1999.

[8] E. Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *Journal of Universal Computer Science*, 2 (8): 2-74, Springer Pub. Co., 2002.

[9] E. Börger, U. Glässer and W. Müller. Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In C. Delgado Kloos and Peter T. Breuer, editors, *Formal Semantics for VHDL*, Kluwer Academic Publishers, 1995, 107-139.

[10] D. Box, *Essential COM*, Addison-Wesley, Reading, MA, 1998.

[11] P. Clements, A Survey of Architecture Description Languages. In Proc. Eighth Intl. Workshop in Software Specification and Design, Paderborn, Germany, March 1996.

[12] E. Christensen et al. Web Service Description Language (WSDL). W3C Note, March 15, 2001, URL: www.w3.org/TR/wsdl.

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

[14] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):97-105, June 2000.

[15] R. Eschbach, U. Glässer, R. Gotzhein, M. von Löwis and A. Prinz. Formal Definition of SDL-2000 – Compiling and Running SDL Specifications as ASM Models. *Journal of Universal Computer Science*, 11 (7): 1025-1050, Springer Pub. Co., 2001.

[16] Foundations of Software Engineering Group at Microsoft, the website, http://research.microsoft.com/fse.

[17] U. Glässer, Y.Gurevich and M. Veanes, Universal Plug and Play Machine Models, Foundations of Software Engineering, Microsoft Research, Redmond, Technical Report, MSR-TR-2001-59, June 15, 2001.

[18] U. Glässer, Y. Gurevich and M. Veanes. High-level Executable Specification of the Universal Plug and Play Architecture.. In Proc. of *35th Hawaii International Conference on System Sciences* (HICSS-35), Software Technology Track, Hawaii, Jan. 2002.

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

[20] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. Springer Lecture Notes in Computer Science 702, 1993, pages 274-308.

[21] Y. Gurevich and J. Huggins: The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. Springer Lecture Notes in Computer Science 1092, 1996, pages 266-290.

[22] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, Oxford University Press, 1995, pages 9-36,

[23] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms, *ACM Transactions on Computational Logic*, vol. 1, no. 1, July 2000, pages 77-111.

[24] Y. Gurevich and N. Tillmann. Partial Updates: Exploration. *Journal of Universal Computer Science*, 11 (7): 917-951, Springer Pub. Co, 2001.

[25] W. Grieskamp, Y. Gurevich, W. Schulte and M. Veanes. Generating Finite State Machines from Abstract State Machines, Microsoft Research, Redmond, Technical Report, MSR-TR-2001-97, Updated May 2002, to appear in *Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis, ISSTA 02.*

[26] D. Hamlet and J. Maybee. The Engineering of Software: Technical Foundations for the Individual. Addison Wesley, 2001.

[27] J. Heering. Application software, domain-specific languages, and language design assistants, in: *Proceedings SSGRR 2000 International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, May 2000.

[28] J. Heering and P. Klint, Semantics of programming languages: A tool-oriented approach, *ACM SIGPLAN Notices*, 35(3):39-48, March 2000.

[29] ITU-T Recommendation Z.100: Languages for Telecommunications Applications - Specification and Description Language (SDL), Annex F: SDL Formal Semantics Definition, International Telecommunication Union, Geneva, 2000.

[30] N. Medvidovic and R.N. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages, *IEEE Transactions on Software Engineering*, 26(1):70-93, January 2000.

[31] R. Stärk, J. Schmid and E. Börger. Java and the Java Virtual Machine: Definition, Verification, Validation. Springer, 2001.

[32] S. Thatte. XLANG: Web Services for Business Process Design. URL: www.gotdotnet.com/team/xml_wsspecs/xlang-c

[33] UPnP Device Architecture V1.0. *Microsoft Universal Plug and Play Summit, Seattle 2000*, Microsoft Corporation, Jan. 2000.

[34] Official Web site of the UPnP Forum. URL: www.upnp.org.

[35] E. Christensen et al. Web Service Description Language (WSDL). W3C Note, March 15, 2001, URL: www.w3.org/TR/wsdl

[36] XLANG Abstract Machine, Foundations Of Software Engineering, Microsoft Research, Internal Report, 2002.