Universal Plug and Play Machine Models

Modeling with Distributed Abstract State Machines

Uwe Glässer
glaesser@upb.de
Heinz Nixdorf Institute, Paderborn, Germany

Margus Veanes
margus@microsoft.com
Microsoft Research, Redmond, USA

Abstract: We present a high-level executable specification for the *Universal Plug and Play* (UPnP) standard illustrating the use of *Abstract State Machine* (ASM) technology as a practical tool for applied systems engineering. The concept of distributed real-time ASM allows us to combine both *synchronous* and *asynchronous* execution models in one uniform model of computation.

Key words: UPnP, ASM, Real-Time Behavior, TCP/IP networking

1. INTRODUCTION

We present here some results of using *Abstract State Machine* (ASM) technology [1] in a recent pilot project at Microsoft. This project was done in collaboration with a group that has developed the *Universal Plug and Play Device Architecture* (UPnP) [2], a distributed, open networking architecture enabling peer-to-peer network connectivity of various intelligent appliances, wireless devices and PCs. UPnP is an evolving industrial standard defined by the UPnP Forum [3].

We have developed a high-level executable specification of the UPnP protocol as basis for the communications software forming the core of Universal Plug and Play Device Architecture. Serving practical needs, we attempt to accurately reflect the abstraction level of the given informal requirements specification [2], where we focus on interoperability aspects rather than on internal details of UPnP components. The construction of a *distributed real-time ASM* allows us to combine both *synchronous* as well as *asynchronous* execution models within one uniform model of computation. We introduce a discrete notion of global system time for dealing with real time behaviour and timing constraints such as delays and timeout events.

From our abstract ASM model of UPnP we have derived an executable version using the *ASM Language* (AsmL) [7]. An additional GUI serves for control and visualization of simulation runs making our abstract executable specification a useful practical tool for the UPnP developers and testers. A comprehensive description is given in [9]. Closely related to this work are ASM behavior models of various programming languages, e.g. Java [12], and modeling languages, e.g. SDL [11] and VHDL [5].[1]

Section 2 illustrates the UPnP protocol, and Section 3 briefly outlines the semantic model used here. Section 4 introduces overall concepts of the abstract protocol model, while Section 5 exemplifies the construction of the device model in some detail. Section 6 contains some concluding remarks.

2. THE UPNP PROTOCOL

UPnP is a layered protocol architecture built on top of TCP/IP networks by combining various standard protocols, e.g. such as DHCP, SSDP, SOAP, GENA, etc. It supports dynamic configuration of some collection of *devices* offering various kinds of *services* requested by *control points*. To perform certain control tasks, a control point needs to know what devices are available, i.e. reachable over the network, what services a device advertises,

[1] An ASM-based formal definition of SDL recently has been approved by the International Telecommunication Union (ITU) as part of the current SDL standard [11].

and when those advertisements will expire. Services interact with entities in the external (physical) world through the actuators and sensors of a device. A sample UPnP device is illustrated in Figure 1.

General Restrictions: In general, the following restrictions apply. Control points and devices interact through exchange of messages over a TCP/IP network, where the specific network characteristics (like bandwidth, dimension, etc.) are left unspecified. Communication is considered to be neither predictable nor reliable, i.e. message transfer over the network is subject to arbitrary and varying delays, and some messages may never arrive. Furthermore, devices may come and go at any time with or without prior notice. Consequently, there is no guarantee that a requested service is available in a given state or that is will become available in a future state. Also, a service that is available need not remain available until a certain control task using this service has been completed.

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 vendor specific information.

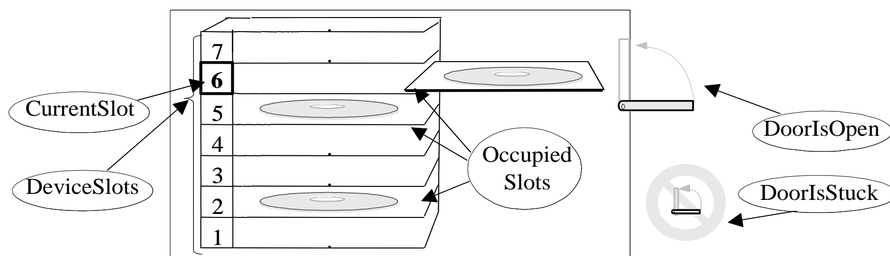

Figure 1. A generic *CD player* as a sample UPnP device. The picture illustrates the state information associated with one of the services, called *ChangeDisc*, associated with the CD player. This service provides functionality to add or remove discs from the CD player, to choose a disc to be placed on the tray, and to toggle (open/close) the door.

3. ABSTRACT STATE MACHINES

This section briefly outlines the model of *distributed real-time ASM* at an intuitive level of understanding and in a rather informal style. For a rigorous mathematical definition, we refer to the theory of ASMs [4,5].

The Basic ASM Model. An abstract machine model *A* is defined over a fixed *vocabulary V*, some finite collection of function names and relation names. Formally, relations are treated as Boolean valued functions, i.e. *predicates*. States of *A* are *structures*[2] defining interpretations of *V* over a common base set. Starting from a given initial state, machine *runs* abstractly model executions of a system under consideration through finite or infinite sequences of state transitions.

The behavior of *A* is defined by its program *P*. Intuitively, *P* consists of one or more state transition *rules* specifying the runs of *A*. Each execution step computes some finite set of local function *updates* over a given state of *A* and fires all these updates simultaneously in one atomic action. We define the rules of *P* inductively as composition of basic *update instructions* by means of simple rule constructors as illustrated in the subsequent sections.

The canonical rule constructor is the **`do in-parallel`**, which allows for the *synchronous* parallel composition of rules. In the below rule, the update set computed by `R` is defined to be the union of the individual update sets as associated with `R1` and `R2` respectively. The '**`do in-parallel`**' is optional (and usually is omitted).

```
R ::= do in-parallel R1 R2
```

Concurrency. A distributed ASM is a generalization of the basic model. It consists of several autonomous ASM *agents* interacting with each other by reading and writing shared locations of global machine states. The underlying semantic model regulates such interactions so that potential conflicts are resolved according to the definition of *partially ordered runs* [4]. Agents come as elements of a dynamically growing and shrinking domain `AGENT`, where each agent has a *program* defining its behavior. The elements from a static domain `PROGRAM` collectively represent the distributed ASM program.

Real Time. Time values are modeled as real numbers by the elements of a linearly ordered domain `Time`. We define the relation "≥" on time values by the corresponding relation on real numbers. Another domain `Duration` represents finite time intervals as differences between time values.

```
domain Time, domain Duration
```

[2] We refer here to the notion of *structure* as it is used in first-order logic.

Global system time, as measured by some discrete clock, is represented by a *monitored*, nullary function `now` taking values in `Time`. A monitored function is an abstract interface to the system environment; as such, it changes its values depending on external actions and events. That is, one can only observe, but not control, how physical time evolves.

```
monitored now : TIME
```

As another integrity constraint on runs, we assume that agents react instantaneously, i.e. they fire a rule as soon as they reach a state in which the rule becomes enabled.

4. ABSTRACT PROTOCOL MODEL

A reasonable choice for the construction of an abstract UPnP model is a distributed real-time ASM consisting of a variable number of concurrently operating and *asynchronously* communicating components. Intuitively, a component either represents a device, a control point or some fraction of the underlying communication network. Components have *interfaces* so that any interaction between a component and any other component is restricted to actions and events as observable at these interfaces.

The external world, i.e. the environment into which the system under consideration is embedded, affects the system behavior in various ways. For instance, the transport of messages over the communication network is subject to arbitrary delays and some messages may never arrive. Also, the system configuration itself changes as components come and go. Those external actions and events are basically unpredictable and as such they are modeled through a GUI allowing for user-controlled interaction with the external world. The overall organization of the model is illustrated in Fig. 2.

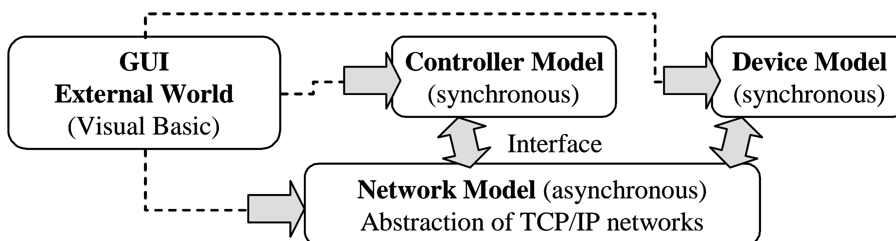

Figure 2. Overall organization of the distributed ASM model of UPnP.

4.1 Components and Interfaces

We formulate behavioral properties of UPnP protocol entities in terms of component interactions, where components are agents of a distributed ASM as identified by a given system configuration.

Conceptually, any interaction between the model and the external world involves two different categories of agents: (1) explicit agents of the model, namely control point agents, device agents or network agents, and (2) implicit agents living in the environment. The non-deterministic nature of environment agents faithfully reflects the system view of the environment.

At the component level, control points and devices are modeled as parallel compositions of *synchronously* operating machine models. Each component further decomposes into a collection of parallel ASMs, one for each protocol phase. In contrast, network components internally are based on an *asynchronous* model with decentralized control as explained below.

Communication Infrastructure. We define an abstraction of TCP/IP networks based on standard network terminology [8]. Our network model is based on a distributed execution model with asynchronous communication according to the view that a complex communication network usually consists of some (not further specified) collection of interconnected physical networks. The network model is described in greater detail in [10].

Transport Protocols. User level processes, or application programs, interact with each other by exchanging messages using the standard transport level protocols UDP and TCP. There may be several application programs running on a single host. Thus the *address* of an application program is given by the IP address of its host in conjunction with a unique protocol port number on this host. In our case, several control point programs may run on the same host. Devices, however, are considered as individual hardware units; therefore they are identified with the hosts on which they run. Collectively, we refer to control points and devices as *applications*.

DHCP Server Interface. The Dynamic Host Configuration Protocol (DHCP) enables automatic configuration of IP addresses when adding a new host to a network. We model interaction between a DHCP server and the DHCP client of a device explicitly only as far as the device side is concerned (cf. Section 5). The server side is implicitly given by one or more external DHCP server agents whose behavior is left abstract. In our model, a DHCP server represents another type of application program.

4.2 Basic Agent Types

We can now define AGENT as a derived domain, where we assume the four underlying domains COMMUNICATOR, CONTROL·POINT, DEVICE and DHCP·SERVER to be pairwise disjoint.

```
AGENT ≡ APPLICATION ∪ COMMUNICATOR
APPLICATION ≡ CONTROL·POINT ∪ DEVICE ∪ DHCP·SERVER
```

Depending on its type, agents either execute the program RunControlPoint, RunDevice, or RunNetwork. The behavior of DHCP server agents is not explicitly defined in terms of a program; rather it is determined by the respective actions of the external world.

```
domain PROGRAM ≡ {RunControlPoint,RunDevice,RunNetwork}
```

4.3 Abstract Data Structures

Mathematical modeling of complex system behavior requires appropriate abstractions for coping with the complexity and diversity of real-life systems. To simplify the formal representation of our model, we assume a rich background structure for *sets* and *maps* with sets of integers, maps from integers to strings, or even sets of such maps, etc. Both maps and sets may be viewed as aggregate entities and may be updated point-wise. We exemplify our approach below.

Messaging. Assume a static universe ADDRESS of *IP addresses* extended by *protocol port numbers* to refer to the global TCP/UDP address space and a unary function address associating with each application some element from ADDRESS. A distinguished address, called thisDevice, serves as a source address for devices that do not yet have an IP address.

```
address : APPLICATION → ADDRESS
```

Messages are uniformly represented as elements of a domain MESSAGE. Each message is of a certain type from the static domain MSG·TYPE. The message type determines whether a message is transmitted using UDP or TCP, though we do not make this distinction explicit. Further, a message identifies a sender, a receiver, and the actual message content, or payload.

An agent has a local mailbox for storing messages until these messages will actively be processed. The mailbox of an application represents its local input port as identified by the respective port number for this application.

```
mailbox : AGENT → Set of MESSAGE initially empty
```

Timeout Events. Agents have several distinct timers for different purposes. Each individual timer t has its own default duration effectively

determining the expiration time when setting `t`. In a given state, a timer `t` is `active` if and only if its expiration time `time(t)` is greater than the value of `now`; otherwise, `t` is called *expired*.

```
duration : AGENT → Map of TIMER·TYPE to DURATION
time : AGENT → Map of TIMER·TYPE to TIME
```

For a given timer `t` of agent `a`, the operation of setting `t` is defined as follows: SetTimer(a,t) ≡ time(a)(t) := now + duration(a)(t).

In a given state, a predicate `Timeout` indicates for given timer instance `t` and agent `a` whether the timer instance is active or has expired.

```
Timeout : AGENT → Map of TIMER·TYPE to BOOL,
Timeout(a,t) = now ≥ time(a)(t)
```

5. DEVICE MODEL

We define the device model as *parallel composition* of six synchronously operating component ASMs, each of which runs a different protocol phase. For illustrating the approach, we restrict here on *Addressing* and refer to [9] for a comprehensive definition of the complementary protocol parts.

Device Status. In a given device state, [2] distinguishes three basically different situations: *inactive*–the device is not connected to a network; *alive*–the device is connected and may remain connected for some time; *byebye*–the device is connected but is about to be removed from the network. The device status is affected by actions and events in the external environment as expressed by an externally controlled function *status* defined on devices.

```
monitored status : DEVICE → { inactive, alive, byebye }
```

In the device model defined below, *me* refers to a device agent performing the program `RunDevice`. The component behavior is defined by the respective ASM rule macros specifying parallel operations of *me*.

```
RunDevice =
  if status(me) ≠ inactive then
    RunAddressing   //Component ASM for Addressing phase
    RunDiscovery    //Component ASM for Discovery phase
    RunDescription  //…
    RunControl
    RunEventing
    RunPresentation
```

Addressing. IP address management requires a DHCP server to assign an IP address when some new device (for which no IP address is specified manually) is added to the network. As reply to a DHCPDISCOVER message from a device's DHCP client, the server broadcasts a DHCPOFFER message identifying the IP address as well as the hardware address of the device.

When no DHCP server is available, a host may obtain a *temporary* IP address through auto IP addressing. This address can then be used until a DHCPOFFER message eventually is received (see [9] for details).

We abstract from any specific algorithms used for auto IP addressing by making a nondeterministic choice to determine a temporary IP address. For checking the validity of a chosen address, i.e. for testing whether this choice causes any conflicts, we assume to have some externally controlled decision procedure as represented by the predicate *ValidAutoIPAdr*.

```
monitored ValidAutoIPAdr : DEVICE × ADDRESS → BOOL
```

In the Addressing ASM below, `RunDHCPclient` models the interaction between the local DHCP client and the DHCP server. The client uses a timer for reissuing its IP address request repeatedly until it eventually receives a response from a server. The first timeout event also triggers the calculation of a temporary IP address in parallel to the execution of the DHCP client.

```
RunAddressing ≡
  if address(me) = thisDevice or AutoConfiguredAdr(me)
    then RunDHCPclient
  if address(me) = thisDevice and ¬DhcpOfferRcvd and
     Timeout(me,dhcpClientTimer) then
    choose adr ∈ ADDRESS: ValidAutoIPAdr(me,adr) do
      address(me):= adr
    AutoConfiguredAdr(me):= true
where
  DhcpOfferRcvd ≡ ∃ m ∈ mailbox(me): DhcpOffer(m)
```

6. CONCLUSIONS

We illustrate here the construction of an abstract operational model of the *Universal Plug and Play* (UPnP) protocol. The concept of distributed real-time ASM allows us to combine synchronous and asynchronous execution models in one uniform model of computation. A notion of global system time allows for dealing with timing constraints.

For a comprehensive description of a fully executable model, including a GUI for control and visualization of simulation runs, sample control points and sample devices, see our technical report [9]. A more specific discussion

on the role of the executable language AsmL as a domain-specific language for rapid prototyping in the UPnP project can be found in [10].

Conceptually, the abstract ASM model complements the informal requirements specification [2] serving as technical documentation, e.g. for further development, whereas the executable Asml model provides a basis for experimental validation, e.g. rapid prototyping and conformance testing.

REFERENCES

1. Abstract State Machine Web site. URL: http://www.eecs.umich.edu/gasm/
2. UPnP Device Architecture V1.0. *Microsoft Universal Plug and Play Summit, Seattle 2000*, Microsoft Corporation, Jan. 2000.
3. Official Web site of the UPnP Forum. URL: http://www.upnp.org
4. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995, 9-36.
5. 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 Pub., 1995, 107-139.
6. Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. ACM Trans. on Computational Logic, 1 (1): 77-111, July 2000.
7. AsmL Web site. URL: http://www.research.microsoft.com/fse/asml/ Foundations of Software Engineering, Microsoft Research.
8. D. E. Comer. Internetworking with TCP/IP, *Principles, Protocols, and Architectures*. Prentice Hall, 2000.
9. 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.
10. 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*, Software Technology Track, IEEE 2002.
11. 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*, 7 (11): 1025-1050, Springer Pub. Co., 2001.
12. R. Stärk, J. Schmid and E. Börger. Java and the Java Virtual Machine: Definition, Verification, Validation. Springer, 2001.