Formal Specification of SNMP MIB's Using Action Semantics: The Routing Proxy Case Study

Elias Procópio Duarte Jr., Martin A. Musicante
Federal University of Paraná, Dept. Informatics
Caixa Postal 19081 Curitiba PR 81531-990
Brazil
{*elias,mam*}*@inf.ufpr.br*

Abstract

The usual way to describe the semantics of MIB objects is just to give an informal English text explaining each object's behavior. Informal descriptions are vague and incomplete. They are open to misinterpretation and may lead to inconsistent implementations. In this work we propose the use of Action Semantics as a simple and powerful tool for the formal description of the behavior of MIB objects. Formal descriptions may be used as the basis for systematic development, verification, and automatic generation of implementations. In our approach, operations for each MIB object can be simply added to the existing ASN.1 MIB descriptions, without implying in any modification of current standards. We initially define the semantics of the core SNMP server, as well as the snmpget and snmpset applications. This description is extensible, allowing the inclusion of any MIB. As a case study, we define the semantics of the experimental SNMP Routing Proxy MIB.

Keywords

Formal Methods, SNMP MIB Semantics, Network Fault Monitoring

1. Introduction

The Internet Standard *Simple Network Management Protocol* (SNMP) Framework defines a *Management Information Base* (MIB) as a collection of related management objects which an *agent* keeps in order to allow management applications to monitor and control the managed entities [7]. The *Structure of Management Information* (SMI) defines the rules for describing management information [6]. The SMI uses a a subset of the *Abstract Syntax Notation One* (ASN.1) language to define MIB modules and objects. In this framework, ASN.1 is also used to define the formats of the packets exchanged by the management protocol.

The current way to describe the *semantics* of SNMP MIB objects is just to give an informal English text explaining each object's behavior. These semantic descriptions can be added either in the form of comments, or in the `DESCRIPTION` fields of SMI definitions. Informal descriptions are usually vague and incomplete. They are open

to misinterpretation and may lead to inconsistent implementations.

In this work, we propose the use of *Action Semantics* [1] as a simple and powerful tool for the formal description of the semantics of MIB objects. Action Semantics is a formal framework for semantic description, developed to provide readable descriptions of real-life languages. Action Semantics descriptions map abstract syntax to semantic entities, which are defined inductively using semantic equations. These equations define *actions* rather than higher-order functions, used in other formalisms, and the essence of actions is much more *computational* than that of pure mathematical functions.

In our approach, actions for each MIB object can be simply added to the existing ASN.1 MIB descriptions, in the DESCRIPTION field, without implying in any modification of current standards. These formal descriptions are precise and allow MIB descriptions to be expressed in a well-defined, established notation. They may be used as the basis for systematic development, verification, and automatic generation of implementations [8, 9]. Furthermore, a formal MIB description supports verification and tractable reasoning about equivalence and other features [10].

Action semantic descriptions are inherently modular. They are easily extended or modified [11]. It is straightforward to reuse parts of the specification. In this paper, we demonstrate these features by creating a core SNMP server specification, which can be extended to include any MIB.

We initially describe the semantic entities that are common to all SNMP MIB's. They include the SNMP server, i.e. the agent that keeps the MIB – and client actions, including the operations get and set, which are used to access MIB objects. Then, as a case study of MIB description, we define the semantics of an experimental MIB: the SNMP routing proxy [3]. This proxy can be used by an application to reach an agent through an alternative path, whenever the route given by the network layer is not working properly. Its deployment has been shown to improve the overall dependability of the network management system [4].

The rest of the paper is organized as follows. Section 2 makes a review of Action Semantics and those features that will be used to describe MIB object's behavior. In section 3 the semantic entities that describe the core SNMP server are defined. In section 4 the Proxy MIB is introduced and then described using Action Semantics. Section 5 concludes the paper.

2. Action Semantics

Action semantics [1] is a formal framework for semantic specification, developed to provide "readable" descriptions of real-life languages [12, 13, 14]. Action semantic descriptions are *compositional*, i.e. they define semantic functions to map abstract syntax objects to semantic entities. Semantic functions are defined inductively using equations. The semantic entities are *actions*, ad-hoc entities which provide a natural way to describe computations.

Action semantics uses a special notation to describe actions. This notation is called *action notation*, and it is used in action semantic descriptions very much in

the same way as the λ-notation is used in denotational semantics [15]. The symbols used in action notation are intentionally verbose, so that English-like phrases can be used—completely formally—to express most of the concepts present in computing.

Action semantics has features that are similar to other semantic formalisms. It is similar to denotational semantics which uses semantic functions to describe the meaning of objects. However, actions have a more operational flavor than functions. In this sense, action semantics bridges the gap between denotational and operational semantics [16].

Actions are used to describe the meaning of computation. Actions can be *performed* to process *information*, with various possible outcomes: normal termination (performance of the action *completes*), exceptional termination (it *escapes*), unsuccessful termination (it *fails*) or non-termination (it *diverges*). Action notation provides some primitive actions, and various *combinators* for forming complex actions, corresponding to the main fundamental concepts of programming languages.

A *data notation* is used to describe the information processed by actions. The standard data notation (included in action notation) provides a collection of algebraically defined abstract data types, including numbers, characters, strings, sets, tuples, maps, etc.; further data may be specified *ad hoc*.

There is also a third class of entities in action notation, called *yielders*. A yielder represents unevaluated data, whose value depends on the current information available to the primitive action in which it occurs. Yielders are *evaluated* to yield data. An example of a standard yielder is the data bound to I, which depends on the current bindings that are received by the enclosing primitive action.

Actions can represent pure control, or can process different types of information. The so-called 'facets' of an action represent the behavior of the action. Each facet deals with one aspect of the information processed by the action. There exists five facets of each action:

Basic: This facet deals with pure control flow, without reference to information processing issues.

Functional: This facet deals with *transient* data, which is given to or by an action. For example, when the primitive action give the successor of the given natural is given a natural number n as transient data, it completes, giving $n + 1$ as a transient. The compound action A_1 then A_2 performs the action A_1 first; all transient data given by A_1 is passed on to A_2, which is performed after A_1 completes. The primitive action choose D, where D is a sort of data, makes a non-deterministic choice of an individual of sort D, giving the chosen datum as a transient.

Declarative: This facet deals with the manipulation of *scoped* information, represented by associations of *tokens* to bindable data. For example, performance of the primitive action bind "max-length" to 256 completes, producing a binding of the token "max-length" to the natural number 256.

Imperative: This facet is concerned with *storage* handling. A storage in action notation is simply a mapping from memory cells to storable data. For example, consider the action allocate a cell then store 26 in the given cell, which combines features of the functional and imperative facets.

Communicative: This facet provides a system of *agents*, which can each be 'contracted' to perform particular actions. Initially only a special 'user' agent is active. Agents can communicate using asynchronous message passing. Each agent has its own *communication buffer*, in which all the messages sent to the agent are placed. Arbitrary data can be contained in messages.

3. SNMP Semantic Entities

In this section, we specify the basic building blocks that will allow any MIB to be formally specified using Action Semantics. The semantic entities defined here are common to all SNMP MIB's. They include the SNMP server, i.e. the agent that keeps the MIB, and client actions, for instance snmpget and snmpset, which are used to access MIB objects.

In the definitions below, equations are followed by comments. The comments are written between horizontal lines.

3.1 Client Side

This part of the specification details the actions used to request an SNMP service, i.e. snmpget and snmpset are specified.

The shape and behavior of snmpget is described in the next two expressions. The functionality (typing) of snmpget is declared first. The snmpget function takes the identification of a station (in which the SNMP server is running), a community (to validate the access to the requested object) and the identification of the requested object. The community is defined to be an item of data belonging to the sort DisplayString. The result of the semantic function is an action, whose behavior is stated by item (1), and it is explained below.

- snmpget(_, _, _) :: IpAddress, DisplayString, OBJECT IDENTIFIER → action

(1) snmpget(A:IpAddress, C:DisplayString, O:OBJECT IDENTIFIER) =
  | send a message[to A][containing ("get", C, O)]
  then
  | accept a message[from A][containing a datum | "wrong community"]
  then
  | give the rest of the given tuple

The semantics of snmpget is stated as follows: a message containing the request is sent to the server (A). The send primitive sends a message to the specified station, A. The contents of the message is detailed in the containing field.

The client will then wait for a reply, which can contain the requested object value, or an error message (in the case a wrong community was given). The symbol " | " means set union. The accept action is defined in section 3.4 below. It waits for a

new message, returning the tuple formed by the sender of the message, followed by the contents of the message.

The basic action give and the combinator then were explained in section 2. They belong to the functional facet of Action Notation. The action give the rest of the given tuple receives the tuple given by the previous accept. This action simply drops the identification of the sender, returning the contents of the last received message (recall the explained behavior of accept above).

The semantics of snmpset below is similar to that of snmpget. In order to set an object's value, a message is issued to the server A. After the attribution of the new value, the server replies to the client a message containing the object identifier and the value set, if the message is accepted.

```
• snmpset(_, _, _, _) :: IpAddress, DisplayString, OBJECT IDENTIFIER, datum → action
(2) snmpset(A:IpAddress, C:DisplayString, O:object-ID, V:object-value) =
        | send a message[to A][containing ("set", C, O, V)]
      then
        | accept a message[from A][containing (O, V) | "wrong community"]
      then
        | give the rest of the given tuple
```

3.2 Server Side

This part of the specification deals with the actions relative to the server side of SNMP. The SNMP-MIB-Daemon action represents the behavior of the server. It is described by a unfolding . . . unfold construction, which is used in Action semantics to represent loops. The performance of an action unfolding A performs the A, but whenever unfold is reached, action A is performed instead. In other words, the dummy action unfold is behaves like a macro, which is replaced by action A itself.

The semantic definition of the server action below treats coming snmpget and snmpset messages in separate actions, given by service-get and service-set respectively. There are other possible approaches to define that server; the present one was chosen for simplicity: all we need for the moment is to reply to client requests.

The _ or _ combinator was used in defining the SNMP-MIB-Daemon action. This combinator represents a choice between alternative actions, in this case, service-get and service-set.

```
• SNMP-MIB-Daemon :: action
(1) SNMP-MIB-Daemon =
      unfolding
        | | | service-get
        | | or
        | | | service-set
        | and unfold
```

Next, the snmpget service is specified. The server receives a message from a client, containing a triple. This triple is formed by the token "get", a community and the object identifier which value is wanted. The action accept returns a tuple, which

will be passed to the action that follows the then combinator. In our case, the tuple given by accept has four elements: a station address (the sender of the message), the token "get", a community and an object identifier.

The action check completes whenever its parameter evaluates to true. In the case of our specification, IS-OK _ will result in true if the community is valid for reading the value of the requested object; otherwise, it results in false. Notice that the components of a tuple are numbered in Action Semantics. The expression the DisplayString#3 refers to the third component of the tuple given to the action, which, in this case must be a community.

After checking the community, the *get* operation can be started. This is performed by the action get(_), which will be defined for each MIB, in the *semantic functions* part of the MIB specification. An object value must be given as the result of each get(_) action.

This combinator and represents the (possibly) interleaved performance of both its component actions. In the functional facet, the and combinator returns the tuple formed by the concatenation of the tuples returned by its component actions.

- service-get :: action

(2) service-get =

```
    accept a message[from any IpAddress][containing ("get", a DisplayString,
                                                        an OBJECT IDENTIFIER)]
then
    check IS-OK(the DisplayString#3, "get", the OBJECT IDENTIFIER#4)
    and then
        give the IpAddress#1 and get(the OBJECT IDENTIFIER#4)
        then send a message[to the IpAddress#1][containing the datum#2]
or
    check not IS-OK(the DisplayString#3, "get", the OBJECT IDENTIFIER#4)
    and then
        send a message[to the IpAddress#1][containing "wrong community"]
```

The specification of service-set is similar to that of service-get. However, there are two main differences between them: (i) The received message must include all the data in the previous case, and contain the new value, to be assigned to the MIB object; and (ii) the operation set(_, _) is used to set the MIB object to the new value.

As in the previous case, the set(_, _) operation must be specified in the semantic functions part of the description. It will receive an object identifier and a value as parameter. It will return the same object identifier and its value.

- service-set :: action

(3) service-set =

```
| accept a message[from any IpAddress][containing (“set”, a DisplayString,
|                                                   an OBJECT IDENTIFIER,
|                                                   a datum)]
then
| | | check IS-OK(the DisplayString#3, “set”, the OBJECT IDENTIFIER#4)
| | and then
| | | give the IpAddress#1 and set(the OBJECT IDENTIFIER#4, the datum#5)
| | | then send a message[to the IpAddress#1][containing rest of the given tuple]
| or
| | | check not IS-OK(the DisplayString#3, “get”, the OBJECT IDENTIFIER#4)
| | and then
| | | send a message[to the IpAddress#1][containing “wrong community”]
```

3.3 Values

We have not included here the detailed formal definition of values, such as display strings, object identifiers and data values. Values can be defined in a similar manner as in [2]. The formal definition of these families of values is straightforward, being just a translation from those defined in the ASN.1 standard.

The notation □ used in the specification below means that the left-hand side of the equation is not defined at this stage.

(1) Simple Type = INTEGER | OCTET STRING | OBJECT IDENTIFIER | BIT STRING
(2) INTEGER = □
(3) OCTET STRING = □
(4) OBJECT IDENTIFIER = □
(5) BIT STRING = □
(6) Application Wide Type = □
(7) Simply Constructed Type = □

3.4 Auxiliary

This section is devoted to the definition of auxiliary semantic entities and actions that are used in the rest of the description.

The accept _ action is defined as follows: (i) a message M is received and (ii) the tuple formed by the identification of the sender of the message and its contents is returned.

- accept _ :: message → action

(1) accept M =
```
    | receive a message
    then
    | give ⟨ the sender of the message, the contents of the mesage ⟩
```

The Action Notation sort agent is used in action semantics to identify the machinery responsible for the performance of an action. The sort agent is loosely defined in [1, Appendix B] as a sub-sort of individual data. In this work we specialize this specification, defining action semantics agent as IP addresses, which is the natural choice in our case.

(2) agent = IpAddress

4. The SNMP Routing Proxy

The standard framework for Internet management is comprised of a network management station (NMS) which communicates with agents using the Simple Network Management Protocol (SNMP) [7]. The NMS queries the agents for management information describing the state of links, devices, protocol entities and nodes. Agents may also send event information to the NMS by using traps. The NMS takes decisions related to fault diagnosis, performance management, and network configuration, among others, based on the collected information.

There is a pressing need for network management systems capable of handling errors. Although network management systems are in principle responsible for fault diagnosis and management, current systems often become partially non-operational as a consequence of the faults they should instead be helping to solve. If a communication link along the path from the NMS to an agent or to a managed network is down, there will be a collapse of network management, as the NMS won't be able to determine the state of part of the managed network.

To make the network management system resilient to network failures there has to be alternative means of accessing agents. The network, in general, possesses multiple potential paths between two communication nodes. However, since network management systems are application layer entities, these have little or no control over the paths that will be chosen by the network layer for routing the management queries. So, alternative paths for management communication have to use application layer entities which relay the management queries and replies along adequate communication routes.

Using the concept of a *proxy*, the NMS has a simple application routing engine to implement a fault tolerant routing system [4, 5]. An SNMP proxy is an entity used by the NMS to access another device, i.e., the proxy receives the query, transmits it to the agent, gets the reply and sends it back to the NMS.

4.1 Application Routes

Consider the simple network topology in figure 1, where the NMS is connected to an agent (Ag) and also to two gateways, G1 and G2. Considering communications involving NMS and Ag, suppose that routing is such that the direct link is used to communicate the queries and replies, as shown in part A of the figure. If the link between the NMS and agent fails, network management queries will be delayed until the network layer recovers from the failure. The delay may be significant as a new route for the agent must be discovered. A proxy could promptly relay the queries from NMS to AG and the corresponding replies from Ag to NMS, as shown in part B of the figure. The condition to obtain this solution is that the routes used by the proxy be available when a failure occurs in the network route between manager and agent. In the example, G2 can be used as proxy in such situation.

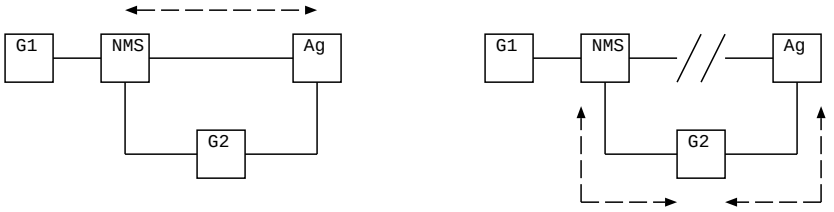

Figure 1: Management communication routes.

An *application route* is a concatenation of one or more *network routes*, which are joined by an application. Thus, the network route from the NMS to the proxy and the network route from the proxy to the agent result in an application route from the NMS to the agent when concatenated. Network routes are not transitive, so if there is a network route from node A to node B, and another network route from node B to node C, the concatenation of these two network routes may be different from the network route from node A to node C. Thus, the application route can be used as an alternative when there is a fault along the corresponding network route.

4.2 Locating Proxies

For a simple network topology like that of figure 1 the position of the proxy is quite obvious, but for a more complex network, like that of figure 2 it is not a simple decision. Considering the scenario where the NMS is attached to Kyoto and there are agents attached to all other nodes. If any network route from the NMS to an agent is not available, the agent will become unreachable to the NMS. A set of proxies should be determined such that whenever an agent becomes unreachable an application route will be established to reach that node.

In [3] an algorithm is presented to determine a set of proxies that allows application routes to be activated whenever network routes between the NMS and an agent is faulty.

For example, consider the network of figure 2. Table 1 gives the results after the candidate proxies are selected based on the number of alternative paths and on the sizes of these paths. For example, for node *ku* there are three possible proxies *tokyo*, *tu*, *tisn*. In the second step the algorithm deals with the selection of one of these three candidates. For each candidate there is a counter of the number of agents for which it is a candidate, the one that may be a proxy for the largest number of agents is selected. In this case, *Tisn* is a candidate for 7 agents; *tokyo* is a candidate for 5 agents; *tu* is a candidate for 6 agents; *tisn* is then selected as proxy for ku. Table 1 also shows *risky nodes*, for which it is not possible to determine an alternative route to reach.

4.3 Dependability Evaluation

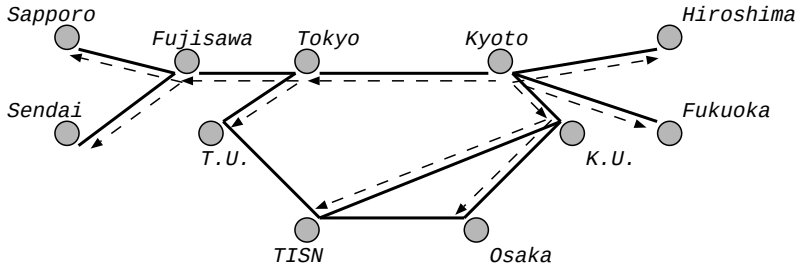

Figure 2: Dotted lines indicate network routes.

Agents	*Candidate Proxies*
ku	tokyo, tu, tisn
sapporo	ku, fujisawa, tokyo, tu, tisn
fujisawa	ku, tokyo, tu, tisn
tohoku.u	ku, fujisawa, tokyo, tu, tisn
tokyo	ku, tu, tisn
tu	ku, tisn
hiroshima	*risky node - no candidates*
fukuoka	*risky node - no candidates*
osaka	tisn
tisn	tokyo, tu

Table 1: Proxies selected for the example network.

To allow an evaluation of the impact of using proxies on network management, we define a measure called *vulnerability*. The *Link Vulnerability*, v_i, for a given link l_i, is the number of nodes that become unreachable to the NMS if l_i is faulty. *Network Vulnerability*, V, for a given network is the summation of link vulnerabilities, for all links in that network, i.e.: $V = \sum_{i=1}^{L} v_i$. The fault coverage, c, of a system gives the probability that the system will recover given the occurrence of a failure in the network. In this context it refers to the probability that the network management system will stay operational if one link fails throughout the network. It can be directly obtained from the previously introduced vulnerability. For the example of figure 2, if one link fails, the probability that the management queries are delivered is approximately 70%. For the network management case, whenever an alternative route exists as an option for the communications that use a given link, the coverage of the system is improved, as the system remains operational.

4.4 Routing Proxy Implementation

The proxy was implemented as a conventional SNMP MIB: a simple and flexible

approach that allows any agent to become a proxy at virtually no cost.

The MIB contains the following objects:

```
agentAD OBJECT-TYPE
        SYNTAX IpAddress
        ...
mgmtOBJ OBJECT-TYPE
        SYNTAX OBJECT IDENTIFIER
        ...
commPXY OBJECT-TYPE
        SYNTAX DisplayString
        ...
resultPXY  OBJECT-TYPE
        SYNTAX DisplayString
        ...
```

Figure 3: SNMP Routing Proxy MIB Objects.

The NMS sets the address of the agent to be queried in variable *agentAD*, the object identifier to be queried in *mgmtOBJ*, and the community that should be used in *commPXY*. After that, by querying the *resultPXY* object, the proxy will issue an snmpget on the agent whose address is *agentAD*, for the object whose identifier is *mgmtOBJ* and using *commPXY* as the community. The result of the query is sent back to the NMS.

4.5 Routing Proxy Specification

The proxy is specified as a conventional SNMP MIB, using the primitive semantic entities given in section 3. This definition is intended to be included in the DESCRIPTION field of the ASN.1 description of the routing proxy MIB.

The action that specifies the routing proxy is Routing-Proxy-Agent below. This action has two main blocks: (i) the initialization of the objects, and (ii) the daemon, which waits for and service incoming messages. This daemon is actually an SNMP agent, described by SNMP-MIB-Daemon (section 3.2).

The routing proxy must allocate memory space for three of its four objects: the three first objects of figure 3. This situation is reflected in the Initialize-Proxy action, where memory cells are allocated to store the values:

- Initialize-Proxy :: action

(1) Initialize-Proxy =
  | allocate a cell then bind “agentAD” to it
  and
  | allocate a cell then bind “commPXY” to it
  and
  | allocate a cell then bind “mgmtOBJ” to it

- Routing-Proxy-Agent :: action

(2) Routing-Proxy-Agent =
    | Initialize-Proxy
    before
    | SNMP-MIB-Daemon

The next equations describe the dynamic behavior of the routing proxy. This behavior is defined as a collection of *get's* and *set's* on the objects of the routing proxy. The three variables "agentAD", "commPXY" and "mgmtOBJ" are set in the functions below.

- set(_, _) :: token, datum $\rightarrow$ action
- get(_) :: token $\rightarrow$ action

(3) set("agentAD", V:IpAddress) =
    | store V in the cell bound to "agentAD"
    then
    | give $\langle$ "agentAD", V $\rangle$

(4) set("commPXY", V:DisplayString) =
    | store V in the cell bound to "commPXY"
    then
    | give $\langle$ "commPXY", V $\rangle$

(5) set("mgmtOBJ", V:OBJECT IDENTIFIER) =
    | store V in the cell bound to "mgmtOBJ"
    then
    | give $\langle$ "mgmtOBJ", V $\rangle$

After the three objects above are set, a *get* can be done on "resultPXY". This operation causes an snmpget to be sent to agent whose IpAddress is stored in "agentAD", asking for the object whose identifier is stored in "mgmtOBJ", using what is stored in "commPXY" as the community. The snmpget operation is defined in section 3.1.

---

(6) get("resultPXY") =
    | snmpget(the IpAddress bound to "agentAD",
    | the DisplayString bound to "commPXY",
    | the OBJECT IDENTIFIER bound to "mgmtOBJ")
    then give the given tuple

---

5. Conclusion

In this paper we have proposed the use of Action Semantics to describe the behavior of SNMP MIB objects. Action Semantics is a formal yet simple tool, that offers many advantages over the current practice of describing MIB objects informally using English sentences. The main advantage of our approach is that it enhances the understanding of MIB semantics.

Action semantic descriptions are inherently modular. They are easily extended or modified [11]. It is straightforward to reuse parts of the specification. In this paper, we demonstrate these features by creating a core SNMP server specification, which can be extended to include any MIB.

The formal descriptions of get and set operations over each object can be simply added to the `DESCRIPTION` field of ASN.1 definitions. There is no need to modify current standards. As a case study we formally describe the experimental SNMP routing proxy MIB. This proxy MIB has been shown to drastically improve the dependability of network management systems [3].

This work is the basis for further experiments on the formal specification of management objects and applications, which include verification, and automatic generation of implementations.

References

[1] P.D. Mosses, *Action Semantics*, Cambridge Tracts in Theoretical Computer Science 26, Cambridge University Press, Cambridge, UK, 1992.

[2] M. A. Musicante, "The Sun RPC Language Semantics," in *Proc. PANEL'92*, Las Palmas de Gran Canaria, Spain, 1992.

[3] E.P. Duarte Jr., *Fault Tolerant Network Monitoring*, Ph.D. Thesis, Dept. Computer Science, Tokyo Institute of Technology, 1997.

[4] E.P. Duarte Jr., G. Mansfield, T. Nanya, and S. Noguchi, "Improving the Dependability of Network Management Systems," *International Journal of Network Management*, to appear in 1998.

[5] E.P. Duarte Jr., G. Mansfield, S. Noguchi, and M. Miyazaki, "Fault-Tolerant Network Management," in *Proc. ISACC'94*, Monterrey, Mexico, 1994.

[6] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser, "Structure of Management Information for Version 2 of the Simple Network Management Protocol", *Request for Comments 1902*, Jan 1996.

[7] M.T. Rose, *The Simple Book - An Introduction to Internet Management*, 2^{nd} ed., Prentice-Hall, Englewood Cliffs, NJ, 1994.

[8] J. Palsberg, *An automatically generated and provably correct compiler for a subset of Ada*, In *ICCL'92, Proc. Fourth IEEE Int. Conf. on Computer Languages, Oakland*, pages 117–126. IEEE, 1992.

[9] D.F. Brown, H. Moura, and D.A.. Watt, *Actress: an action semantics directed compiler generator*, In *CC'92, Proc. 4th Int. Conf. on Compiler Construction, Padeborn*, volume 641 of *Lecture Notes in Computer Science*, pages 95–109. Springer Verlag, 1992.

[10] M.A. Musicante, *On the Relational Semantics of Interleaving Constructors*, Ph.D. Thesis, Dept. Computer Science, Federal University of Pernambuco, Brazil, 1996.

[11] P.D. Mosses and M.A. Musicante. *An action semantics for ML concurrency primitives*, In *Proc. FME'94 (Formal Methods Europe, Symposium on Industrial Benefits of Formal Methods)*, number 873 in Lecture Notes in Computer Science, Barcelona, Spain, October 1994. FME, Springer-Verlag.

[12] J.U. Toft. *Feasibility of using RSL as the specification language for the ANDF formal specification*, Technical Report 202104/RPT/12, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1993.

[13] J.P. Nielsen and J.U. Toft. *Formal specification of ANDF, existing subset*, Technical Report 202104/RPT/19, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1994.

[14] B.S. Hansen and J.U. Toft. *The formal specification of ANDF, an application of action semantics*, In Peter D. Mosses, editor, *Proc. First Intl. Workshop on Action Semantics (Edinburgh, April 1994)*, number NS-94-1 in BRICS Notes Series, pages 34–42. BRICS, Dept. of Computer Science, Univ. of Aarhus, Denmark, 1994.

[15] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn & Bacon, 1986.

[16] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing Series. MIT Press, 1993.