A Methodology for Formalizing GDMO Behavior Descriptions

P. Hasselmeyer
Information Technology Transfer Office
Darmstadt University of Technology
Wilhelminenstr. 7
D-64283 Darmstadt, Germany
peer@ito.tu-darmstadt.de

Abstract

Network management is a key technology for operating large heterogeneous data transmission networks. To allow deployment of equipment from different vendors, the OSI TMN (Telecommunications Management Network) framework defines the language GDMO (Guidelines for the Definition of Managed Objects). Unfortunately, the behavior of managed objects is defined in an informal manner using natural language. This results in behavior specifications which are often vague and ambiguous, increasing the possibility of different implementations not being interoperable. To achieve consistent, clear, concise, and unambiguous specifications, a formal methodology has to be utilized. This paper introduces a framework for the inclusion of formal behavior descriptions into GDMO specifications. An object-oriented logic programming language is presented, which can be used in conjunction with the framework to specify the behavior of managed objects. The language is aimed at automatically producing prototypes of the described system. It enforces strict type checking at compile time to catch errors as early as possible. Furthermore, it works on a rather abstract level to hide specific implementation details.

Keywords

Behavior specification, OSI TMN, GDMO, information models

1 Introduction

Telecommunications traffic is growing constantly. The convergence of telephony and data transmission as well as the increasing use of the World-Wide Web and other Internet services call for a drastic expansion of the telecommunications infrastructure. In this competitive market, management of the network becomes a crucial issue in providing and maintaining a high level of service quality.

Most networks have been growing for a number of years and contain network elements from different vendors. To provide a unified network management of these heterogeneous systems, ISO (International Organization for Standardization) and CCITT/ITU (ITU: International Telecommunications Union, formerly known as CCITT: International Telegraph and Telephone Consultative Committee) introduced

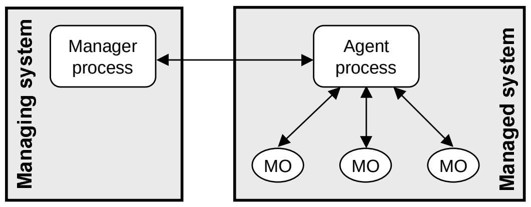

Figure 1: Management model

the TMN (Telecommunications Management Network) framework. The framework considers the management of telecommunications networks as a distributed application. It distinguishes two different roles that a system can play – the manager and the agent role (see Figure 1). Usually, each network element contains an agent process which handles its managed objects under the control of one or more managers. *Managed objects* are used to abstract from the implementation of a device. They can represent physical (e.g. a switch matrix) or logical (e.g. a connection) resources. Even though the hardware platforms of two switches might be completely different, their appearance at the management interface is more or less the same and does not reveal the switch's internal structure.

Managed objects are modeled by attributes, actions, and notifications. Attributes represent a certain part of the internal state of an element. They can usually be read and sometimes be modified. Actions invoke certain functions which a device can perform. Notifications are spontaneous messages emitted if certain events occur. Managed objects which are alike are grouped together to form managed object classes. Classes can inherit their appearance from other classes and add new features.

To enable interaction of managers and agents, the appearance of managed objects can be formally described using the language *GDMO* (Guidelines for the Definition of Managed Objects) [11]. The language defines a number of so-called *templates*. Each template describes a certain aspect of a managed object class (e.g. an attribute) or the class itself. As GDMO does not offer possibilities to define data types, ASN.1 [9] is employed. It is important to note that not only attributes have certain data types – actions and notifications also transfer data which is typed.

Besides the powerful set of notational tools to describe the appearance of managed objects, GDMO offers the *BEHAVIOUR*-template. As the name suggests, it is aimed at describing the behavior of some part of the GDMO specification. In practice, the template often contains additional kinds of information besides the behavior, e.g. about the intended use of an object. Unfortunately, the descriptions contained in these templates are usually plain, informal (English) text. Using informal descriptions has two major disadvantages compared to using some formal notation. First, they are not automatically translatable. Second, by the very nature of human languages, informal descriptions are often vague and ambiguous. Furthermore, experience showed that behavior descriptions are often incomplete. These deficiencies make it impossible to automatically generate prototypes from informal behavior descriptions. Even manual implementations often differ in details because of the different understandings of the descriptions.

To improve the quality of the descriptions and the resulting implementations, a formal method for specifying behavior is desirable. Such a method would make the descriptions unambiguous and thus remove the possibility of misunderstandings. Furthermore, a higher degree of completeness is anticipated as missing features become more readily obvious in a formal, structured specification. Formal behavior descriptions thus make it easier for an engineer to understand the complete information model and to derive a valid, consistent, and compatible implementation from it. Depending on the formal method chosen, it may even enable the automatic generation of simulators or prototypes.

This paper focuses on a framework and a language for formalizing behavior descriptions and combining them with existing GDMO definitions. Special attention is paid to the possibility of automatically generating simulators for given information models. The language has been developed within the MOON (Management Of Optical Networks) project. This project is part of the ACTS (Advanced Communications Technologies and Services) Programme funded by the European Union. A number of leading telecommunications equipment manufacturers and network operators collaborate with research institutions on the development of a platform-independent management framework for the photonic layer of the future pan-European transport network. As described in [6], the special characteristics of all-optical WDM (Wavelength Division Multiplexing) networks are to be identified and described in a management information model using GDMO. Appropriate managers and agents are to be developed and tested. For testing purposes, a simulator software package exists, which implements information model processing facilities as well as a protocol stack for communication. It can process GDMO definitions and retrieve all information about the external appearance of managed objects from it. Behavior of the objects has to be coded manually using a proprietary logic programming language called *RDL* (Rules Definition Language) [8]. This language works on a rather low level of abstraction and does not contain object-oriented features. Analysis showed that this language is not well suited for formalizing behavior descriptions. It was therefore decided to develop a different language which avoids the shortcomings of RDL and is suitable for generally formalizing behavior descriptions in information models. The result of the research is the language which is described in this paper. It is called *BSL* (Behavior Specification Language). This language supports the object-oriented features of GDMO and operates at a high level of abstraction. It hides the internal implementation of the language from the programmer and leads to short and precise specifications. Unlike the language described in [2], BSL is a logic programming language which will be shown to be more appropriate for simulators. A compiler for translating BSL to RDL code has been developed and successfully used within the MOON project.

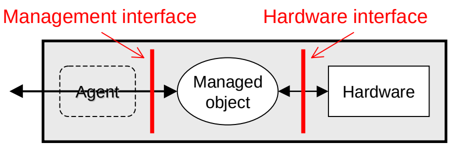

Figure 2: Managed objects as mediators

2 Types of Behavior to be Formalized

Depending on the examined part of a distributed system, different aspects can be identified, which need to be formalized. Looking at the manager, it might be important to identify its reactions to a single or a series of incoming notifications. As these reactions might be different for various network operating companies and are a key factor in the network management procedures, behavior of managers should not be prescribed.

But even when looking at the agent, a number of different aspects can be investigated. An important point is the distinction between behavior which can be observed at the management interface and behavior related to the hardware implementation of a network element. The latter depends on the actual realization of a device and should not be formalized as it would constrain the range of possible implementations. Behavior at the management interface abstracts from the actual implementation of a device and should be consistent across all realizations. It should therefore be described in a formal manner. While the reactions of managed objects to changes at the management interface can be completely formally described, influences on the hardware must be individually identified and adapted for every real device.

Although the agent process is responsible for processing management requests and operating the hardware, managed objects can be viewed as mediators between the network management interface and the hardware (see Figure 2). As these, the general structure of managed objects' behavior is simple: they react to stimuli coming from either side. The reactions might include output to one or even both sides. Typical input from the hardware side includes changed values and timer expirations. Output to the hardware usually consists of setting a number of registers. As the hardware interface should not be formalized, a method for abstracting from the hardware must be found. For input from the hardware it is a natural choice to introduce the notion of *events*. As used in this paper, an event is a certain state of the device, which may be the cause of a reaction. While this reaction is part of an object's behavior and must be formalized, the mechanism for detecting its source is left to the hardware designer. Hardware events which do not change the appearance of the device at the management interface are not formalized, even if they require changes to the hardware. This is considered to be device-internal and might be solved differently in various devices. As incoming management requests must be received by some type of hardware, they can be interpreted as events as well, thus unifying all possible sources of input under one single notion. Management operations can be split up

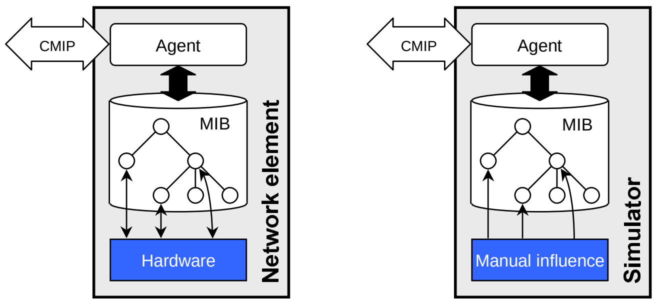

Figure 3: Network element versus simulator

into the events create, delete, action, set, and get.

Another important distinction is to be made between real implementations and simulators. Simulators have two main fields of application. First, a management information model can be tested even though the real device does not yet exist. Mistakes in the information model can be found at an early stage during the development process. Second, a manager can be tested although the managed device is not available. But even if the managed device is available, a simulator can help check the reactions of a manager in rare circumstances which might be difficult to induce on a real device.

As shown in Figure 3, while actual implementations posses both the network management and the hardware interface, simulators lack the hardware and only offer the management interface. Effects of management operations on managed objects can be perfectly simulated but effects on the hardware are lost and events from the hardware can never occur. To be able to simulate at least part of the missing hardware, a possibility to induce events manually or automatically must be given.

The remaining part of this paper aims at developing a framework and a language for formalizing behavior for a simulator. It thus only uses the notion of events but does not describe how to define them. As the simulator possesses the management interface, incoming network management operations generate the appropriate events create, delete, action, set, and get. As these events are available in every TMN agent, they are an inherent part of the framework and thus predefined.

3 Including Formal Behavior in GDMO Descriptions

Before considering a concrete language for specifying managed object's behavior, a methodology for combining formal descriptions with GDMO definitions is presented. The methodology is independent of the language used and can be combined with other approaches for formalizing behavior. It is based on the notion of events and supports the object-oriented features of GDMO.

When combining behavior descriptions and GDMO definitions, two extremes can be identified. The first is to keep both parts separate only linking them by specifying which behavior applies to which objects. The other extreme is to have the behavior included in the GDMO definitions. This can be accomplished in at least two ways. One is introduced in GDMO+ [12] which defines a number of new templates that contain certain aspects of the behavior. The other solution is the inclusion of formal behavior descriptions in regular BEHAVIOUR-templates. These usually contain informal text but can be enhanced by including formal descriptions. In this case, the informal and the formal parts must be distinguishable. An easy solution is the separation by keywords. The beginning and the end of the formal part might be marked by the words “FormalBehaviourBegin” and “FormalBehaviourEnd”. Everything in between is the formal description of some aspect of the behavior. The collection of all aspects within an information model constitutes the complete behavior of a device. The latter method has the advantage of not introducing any new templates. Existing GDMO tools can still be used as they do not know about the formal extension.

The applicability of behavior can be deduced from the placement of its description. BEHAVIOUR-templates are referenced by other parts of the GDMO description directly or indirectly (e.g. via packages or by inheritance). The references imply that the described behavior applies to the referring component. GDMO allows different types of templates to reference the same behavior, e.g. an attribute might show the same behavior as an object. In practice, this feature is not used. It is not explicitly considered in this paper but does not affect the usability of the proposed framework.

Depending on the referring component, the event which triggers a certain behavior might be predetermined. If an ACTION-template references a behavior, the triggering event is already known: it is the reception of that action. On the other hand, if a package references a behavior, no further information is known about the triggering event. Only the classes to which this behavior applies can be deduced. To find out to which event the behavior really applies, some further information must be given inside the formal description. A possible solution is presented in Section 5.1.

The experience with existing information models showed that behavior is usually described in a distributed manner. This is useful if the descriptions are placed according to their semantic meaning in the way described above. Unfortunately, behavior aspects which belong together are often separated and placed in multiple BEHAVIOUR-templates. This increases the difficulty of developing a valid implementation of an information model as all behavior aspects of a component have to be collected and assembled first. The framework described in this paper allows such broad distributions but it requires the specification of the context of each part. This enables the automatic collection of all behavior aspects related to one GDMO component.

4 Important Aspects for Formalizing Behavior

At present, implementation of the management interface usually consists of two steps. First, the object model, described in GDMO, is automatically translated into a programming language. The next step requires the manual insertion of code which implements the behavior described in the informal part of the GDMO specification. For both steps, the same programming language is used. Usually, a general purpose language, e.g. C++, is chosen for this activity. Employing a general purpose language is useful because programmers do not need to learn another programming language. If the language is used for a special application area, developers have to get used to the specific programming environment which prescribes a certain programming style and offers specialized functions. For a limited application area, such as network management, it might be a good choice to employ a special language [3]. Such a language incorporates specific features which are commonly used within its application area. In a general purpose language these features might require a large amount of coding which consumes valuable time and increases the possibility of errors. With a specialized language it is thus possible to quickly generate fully functional prototypes and even complete products.

The design of a programming language for a special application area poses a number of questions. Among other options, a programming paradigm has to be chosen, the type system, and internal operators and functions have to be found. In the next sections two important aspects are discussed and the solutions chosen for the language described in this paper are presented. It is important to note that the language is designed especially for the use within information models. A basic principle of the OSI network management architecture is the abstraction from implementation details. The proposed language tries to adhere to the same goal.

4.1 Programming Paradigm

In the context of network management, a simulation tool can be used for two purposes. The first is to emulate a network resource. The other is to serve as a reference implementation. For both purposes it is useful if the simulator can generate all possible valid responses in reaction to a management operation. For each invocation of the same management operation a different answer might be sent back. When emulating a network resource, this is useful to verify that a manager can cope with different possible results. If employed as a reference implementation, the device must be able to allow, recognize, and generate all possible valid results.

The aspect that there is usually a whole set of correct answers plays an important role in the choice of a suitable language for formal behavior descriptions. The most common programming languages like C and Pascal are based on the imperative programming paradigm. In this domain, a program precisely describes one way to get from the input to the desired output. The implemented way realizes the chosen algorithm. In the domain of logic programming languages a program does not describe a certain way to reach results. A program rather contains a set of facts and a set of rules which state relations among the facts, the input, and the output. Given a certain request, the computer tries to deduce the answer by itself from the given rules and facts [1].

A reasonable application of a reference implementation could be the use as a "judge". Given the device's internal state, a management operation, and the result of another device, the reference implementation would try to deduce that result from the input. If that is possible, the device's answer is correct. If the deduction fails, the answer is incorrect and the device made a mistake. This application area can be covered by a logic programming language very easily. Instead of letting the computer find a solution to a management operation, the answer is supplied as well. The language's search algorithm is now automatically guided by the answer either showing it a valid path or leading to a contradiction.

The previous remarks should have made clear why a logic programming language was chosen for formalizing behavior descriptions. It is more important to describe the valid set of object states, management operations, and responses than to specify an algorithm to calculate only one valid result from some input state and operation. The seamless inclusion of a logic programming language into the TMN framework is easy when thinking of the MIB (management information base) as the set of facts. The incoming management operation is the query and the rules state relationships among the facts, the query, and the result.

4.2 Type System

The type system is an important feature of a programming language. One can distinguish between compilers with static and compilers with dynamic type checking [7]. A compiler with static type checking knows the types of all components (variables, etc.) at compile time. It is therefore not necessary to check the type consistency at run-time, which compilers using dynamic type checking have to do. For some languages both kinds of type checking are possible – at least to a large extent. Other languages only allow dynamic type checking. Usually, compilers working with static type checking offer better performance at run-time. A more important advantage of static type checking is the possibility of finding type errors at an early stage during the development process. Experience with a language only allowing dynamic type checking showed that most programming errors were type errors. When employing static type checking, these errors can be caught at compile-time removing the turn-around time between compiling and executing. The language discussed in this paper thus offers the possibility of static type checking.

A type system within a language used in the area of OSI network management must include at least two kinds of types. ASN.1 types are intrinsically needed, since attributes have these types. The second kind are GDMO types. In line with the object-oriented programming paradigm, managed object classes are considered to be types. To be able to reference objects in the MIB, the language must offer the possibility to store these references in variables. To enable static type checking and allow access to attributes of objects, these references must contain information about the type of the referenced object. It is therefore necessary to define variables which have managed object classes as their types.

As GDMO incorporates inheritance, variables referencing managed objects should support polymorphism [5]. Variables of this kind are allowed to reference managed objects of their own class as well as objects of classes which are derived from that class. Additional attributes of the derived class are not accessible directly as the compiler does not know the actual class of the object referenced at run-time. If an additional attribute is to be accessed, the compiler has to be informed about the actual type of the object. This operator is especially important as attributes which reference objects within the MIB usually have the ASN.1 type “ObjectInstance”. The type of the referenced object is unknown at compile time. Usually only a limited number of classes is supposed to be referenced, though. If the class of the referenced object is fixed, this knowledge can be made available to the compiler which in turn grants access to the class’ attributes. At run-time it has to be ensured that the referenced object has the anticipated type. This is the only situation in which dynamic type checking has to be employed.

5 Behavior Description Language

After describing a number of important aspects which have to be considered when designing a language for behavior descriptions, this section focuses on the syntax and semantics of the language BSL which is discussed in this paper.

5.1 Program Structure

A BSL program consists of two parts. The first declares functions and the context to which the code applies. The second is the code itself.

The important idea in this section is the notion of *contexts*. As described in Section 3, the placement of a behavior aspect can imply its applicability. Often, this is not the case, though, and some additional information is needed to fully determine the behavior’s area of application. The encoding of this information is done by so-called contexts. Each context consists of three elements: the managed object class, the event, and some additional information. The meaning of the additional information depends on the event. For the predefined events, this is listed below:

Event	*Additional Information*
create/delete	–
get/set	attribute’s name
action	action’s name
notification	notification’s name

It is allowed for one or even more elements of a context to be unknown. The unknown elements are simply excluded from the decision about the applicability of the related behavior. This means that if the additional information for an “action”-event is missing, the related behavior is applied to all actions of the respective class (if the class is not known either, the behavior is applied to all actions of all classes). Such underdetermined contexts appear naturally when referencing behavior by attributes or packages. The scope of the behavior can be narrowed by supplying the missing context elements, e.g. the behavior of an attribute can be constrained to just the “set”-event by the BSL code fragment “`context event=set`”.

It is worth noting that adjusting the class in a context contradicts the object-oriented paradigm and is therefore not encouraged. The possibility has been introduced for practical reasons. It eases the restriction of a type of behavior to just a single class, if it is referenced by multiple classes. Another advantage is the possibility of specifying behavior separate from the GDMO definition. This is important when using information models from external sources, e.g. from the ITU-standard M.3100 [4]. If formal behavior descriptions are inserted in such a description, updates of the model require a complete reinsertion of the code. If the model and the behavior are kept apart, each one of them can change without affecting the other.

BSL allows adjusting the context by the language construct “`context` <tag>`=` <value>”. <tag> can be one of “`class`”, “`event`”, and “`info`”. Context modifications can be connected by semicolons.
Example: `context class=software; event=get; info=version;`

The actual program code of a behavior description is preceded by the word “`begin`” and followed by the word “`end`”. Code which is encapsulated by these two words is called a *block*. Blocks can be nested. Each block starts with the declaration of its variables.

5.2 Variables

Variables in logic programming languages are not representing memory locations as in imperative languages. They rather denote a local name for a data object. Values are not assigned to variables – variables are bound to data elements.

Variables in BSL have a certain type and must be declared before use. Declarations can only appear at the beginning of each block and are introduced by the word “`var`”. To be able to use components of ASN.1 data structures as type identifiers, BSL contains the two operators “`.`” and “`elem()`”. “`.`” grants access to components of a choice, a set, or a sequence. “`elem()`” selects the base type of a set-of/sequence-of type.
Example: `var i:Integer; obj:fabric; e:elem(VSet).x`

BSL allows the use of symbolic names wherever possible. It not only allows to access attributes and elements of sets and sequences by their names, it also lets the programmer use the symbolic names for values in enumerations and for types in choices.

5.3 Program Sequences

BSL programs consist of a number of statements. They can be combined by logically or-ing or and-ing them together. For a program to terminate correctly, all and-combined statements must be satisfied. For or-combined statements only one of them must lead to an answer. The order of execution of the statements depends on the type of combination. And-connected statements are executed in the order in which they were written. Or-connected statements allow non-determinism to be introduced. One of those statements is selected at random. If it leads to a contradiction, another possibility is tried until an answer is found. Grouping statements together by “and” is done using the comma (“`,`”), by “or” using the pipe symbol (“`|`”). Following common mathematical rules, “and” has higher precedence than “or”.

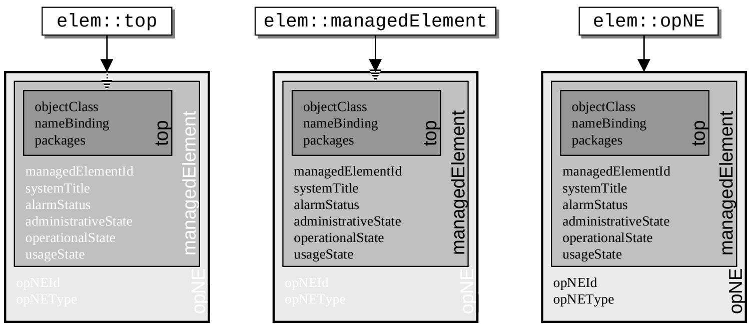

Figure 4: Visibility of attributes

In BSL, each statement and therefore each sequence of statements has a type. This feature is especially needed to determine the return type of functions. The type of a sequence of "and"-connected statements is the type of the last statement. The types of "or"-combined statements must all be the same as it is not known until run-time which alternative is chosen but type checking already occurs at compile time.

5.4 Control Structures

The language BSL incorporates a small number of execution control structures. Two of them allow the selective execution of different program code. These are the "`if-then-else`" statement and the "`case`" statement. It has to be observed that all possible branches must have the same type to ensure static type checking.

Examples:
```
if x=0                choice x of
then y=22                case 0:  y=22
else y=99                case 1:  y=99
endif                    otherwise:
                      y=x*11
                      endchoice
```

The third control structure is an iteration construct. It allows the application of a piece of code to all elements in a set (ASN.1 types set-of/sequence-of).

Example:
```
foreach element in ConnectInfo do
    Print(element.administrativeState)
```

5.5 Operators

BSL offers a number of operators. Besides the usual arithmetic, logic, and comparison operators, it has a few for working with structured data. Especially noteworthy are the operators described in the following sections.

The first operator is the type change operator "`::`" whose functionality has been described in Section 4.2. An example is shown in Figure 4. Supposed that the variable `elem` has the type `top` but references an object of type `opNE`, the usage of the operator "`::`" can make additional attributes visible.

It is important to state a few remarks about the assignment operator "`:=`". Normal variables adhere to the logic programming paradigm and can only be bound to other data items. This is done by the binding operator "`=`" which also serves as the equality operator. It returns the logic value true, if binding was successful or the equality relation holds. Unlike variables, attributes have an imperative semantics. They can be changed at any time and as often as needed using the assignment operator.

The choice of operators for accessing structured data is led by the possibilities of GDMO and ASN.1. Especially the access operator "`.`" is indispensable. Depending on the type of data it is applied to, it allows to read the value of an object's attribute or to select a component of an ASN.1 choice, set, or sequence. Supposed the variable `obj` references an object with the attribute `ConnInf` which has an ASN.1 sequence type containing the choice `itemType` with the element `bidir`. To access that component, one simply writes:
`obj.ConnInf.itemType.bidir`.

To find out which alternative of a choice is present in an instance of that data type, the operator "`type()`" can be used. To get a reference to the object which is denoted by a value of type `ObjectInstance`, the operator "`object()`" is available.

5.6 Functions

BSL allows the definition and invocation of functions. Functions can be declared before the start of the program code. All functions can be accessed globally. Function definitions include the function's name, the declaration of its parameters, possibly a return type, and the code of the function. Each declaration is introduced by the keyword "`function`". Following the keyword "`returns`" the return type of the function can be specified. It is important to note that the return type of the code must match this type.

Example:
```
function mult(Integer a, Integer b) returns Integer :-
        a * b;
```

The availability of a return type might be surprising in a logic programming language. Parameters in these languages can be used for input as well as output. The data direction is determined by the actual arguments in the function call. If an actual parameter is bound to a constant, it is an input parameter, otherwise it is used for returning a value. There are two reasons for a return type being available. First, it allows the explicit specification of an argument as an output parameter making the intended use of the function more obvious. Second, it should ease the semi-automatic translation to an imperative language.

Function invocation is straightforward: the function's name and its actual arguments have to be specified, e.g. `x = mult(3,5)`. BSL incorporates a number of predefined functions. These are designed to facilitate common jobs in a network management environment. There are functions to create and delete objects, handle set types, etc.

6 Conclusion

This paper presented a language for formalizing behavior descriptions in the OSI telecommunications management network framework. A number of questions which arise when designing a language have been discussed and a general framework for the inclusion of formal behavior specifications in GDMO definitions has been presented.

The proposed language was used to formally specify the behavior of the MOON information model as well as of the underlying model M.3100. A compiler for the proposed language has been implemented and used to translate the BSL specification to the language RDL used by the simulation environment. This demonstrated that BSL is capable of specifying the behavior of a reasonably sized information model. A large amount of the behavior could be described in a surprisingly short and easy to understand manner.

The specification of the MOON information model showed that a large part of the behavior was specified in a rather imperative fashion. This gives rise to the possibility of compilers for destination languages other than RDL. Especially common imperative, object-oriented languages like C++ are an important target. Unfortunately, programming languages cannot be translated easily into languages of a different programming paradigm. Specifying behavior as imperative as possible, a translator needing only a minimal amount of manual adaptation seems feasible.

References

[1] Ambler, A. Burnett, M. Zimmerman, B. Operational Versus Definitional: A Perspective on Programming Paradigms. *IEEE Computer 25*, 9 (September 1992), 28-43.

[2] Eberhardt, R., Mazziotta, S., and Sidou, D. Design and testing of information models in a virtual environment. *The Fifth IFIP/IEEE International Symposium on Integrated Network Management "Integrated Management in a Virtual World"*. San Diego, May 1997.

[3] Glass, R. L. How Best to Provide the Services IS Programmers Need. *Communications of the ACM 40*,12 (December 1997), 17-19.

[4] ITU-T. *Recommendation M.3100, Maintenance – Telecommunications management network – Generic network information model*. ITU-T, 1995.

[5] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, 1997.

[6] Mader, W. *D0103: Progress Report 2*. MOON document A231.P01.010.DS.P.010.a1, 1998.

[7] Pratt, T. W. *Programming Languages*. Prentice Hall, Englewood Cliffs, 1984.

[8] Telenet GmbH. *TSE-P-Handbuch*. Telenet GmbH, Darmstadt, Germany, 1997.

[9] CCITT. *Recommendation X.208, Specification of Abstract Syntax Notation One (ASN.1)*. CCITT, 1988.

[10] CCITT. *Recommendation X.701, Information technology – Open Systems Interconnection – Systems management overview*. CCITT, 1992.

[11] CCITT. *Recommendation X.722, Information technology – Open Systems Interconnection – Structure of management information: Guidelines for the definition of managed objects*. CCITT, 1992.

[12] ITU-T. *Draft of ITU-T Rec. X.722/Amd. 4, Information technology – Open Systems Interconnection – Structure of management information: Guidelines for the definition of managed objects – Amendment 4: An extension of GDMO for specifying managed objects behaviour*. ITU-T, 1996.