A Generic and Executable Model for the Specification and Validation of Distributed Behaviors.

Dominique Sidou.
Institut Eurécom,
2229 route des crètes, B.P. 193,
06904 SOPHIA ANTIPOLIS CEDEX, France.
email: sidou@eurecom.fr

February 27, 1997

Abstract

An executable model based approach to validate distributed behavior specifications is presented. Behaviors are simple rewrite rules on data instantiated according to a given information model. They are triggered by events such as operations invoked on computational interfaces. The model is generic in the sense that it does not make any assumption about the information and computation models used, i.e. frameworks such as the OSI Systems Management Architecture, CORBA or SNMP can be adapted. To ensure usability, the execution environment provides for (i) a visual operational semantics of the proposed behavior specification language, and (ii) for dynamic debugging facilities. In addition, a *Scheme* programming environment is shown to be a valuable option to support the implementation, in particular for dynamic debugging purposes.

Keywords : Distributed Behaviors, Data Oriented Executable Specification, Validation, Visual Operational Semantics, Dynamic Debugging, *Scheme*, ODP.

1 Context and Objectives

The paper describes a generic specification and validation framework for distributed behaviors. A special emphasis is put on the genericity and executability of the model, its design and its realization in *Scheme* [4]. Distributed Processing Environments (DPE) have become an available distribution technology to support current and future distributed applications. Satisfactory solutions exist for communication oriented aspects based on protocols, services, and on top Interface Definition Languages (IDL) (e.g. RPC systems, CORBA systems, OSI systems management, SNMP framework). However, behavior aspects are still poorly considered. This inevitably results as inter-working problems between clients and servers. That is the reason why formal approaches are being envisioned, in particular in the ODP framework [16]. First attempts are being made to try to instantiate this framework, e.g. ITU-USG15 [9] work on ODP and TMN information models. Another work of interest is the analysis and design task force [14] launched by the OMG. The proposed model for distributed behaviors aims at a working and usable specification and validation environment. Therefore, the chosen approach for validation is based on executable specifications [8], onto which test cases or scenarios can be exercised. This provides for immediate validation, and can be used directly with the users to confirm that the specification satisfies their initial requirements[1]. Thus as in [20, 1] the objective is quite limited to provide better behavior specifications validated with respect to the informally given original user requirements about a system. As a consequence formal procurement of implementations through proved consecutive refinements steps is not considered.

A model oriented approach is used, that typically describes a system in terms of data and operations. Such approaches have also been qualified as Data Oriented Models (DOM) [13], in opposition to Process Oriented Models (POM), i.e. Process Algebraic notations (CCS) or CSP and FSM based approaches.

[1]In fact, at the point of setting up a first formal specification from the user requirements, executability and prototyping is probably a more usable approach in comparison to e.g. proof systems.

There is, of course, a direct correspondence between DOMs and POMs : both define an abstract machine, i.e. a state and the possible transitions between states. The difference is a matter of presentation. A DOM defines for each action what changes upon state can be expected, whereas, a POM gives for each state what actions are possible. POMs have been used to check properties about the behavior of the whole system (safety and liveness). A DOM seems more adapted to the validation of statements about individual operations, e.g. using *pre* and *post* assertional conditions. However, it is quite simple (and not new [13]) to merge the two approaches, e.g. a DOM with an interleaving semantics. The model advocated follows this approach : it enables to specify guarded action behaviors extended with assertions, that exercise themselves on a notion of Distributed and Shared Memory (DSM). The execution model handles nondeterminism by interleaving the execution of actions. No constraint is imposed about behavior execution contexts. They may include reference(s) to the piece of the DSM concerned with the execution of a behavior. Of course, overlapping can occur which result as potential interference between behaviors. Various notions of execution contexts can be incorporated to the generic kernel by provisioning it with a DSM model and API.

"Ideal" ODP Perspective : The DSM-API corresponds to simple basic primitives to manipulate information objects consistently in regard to the underlying Information Model (IM) used. Information viewpoint languages allows to specify and organize the data in many ways : flat variables, objects, object clusters, relationships.... Note that, the DSM-API can be viewed as a notion of interface to information objects. In contrast, at the Computational level signatures of operations are described using interface specification languages. Operation signatures describe the service primitives that are actually invoked from clients to servers in the distributed environment. ODP recommends some degree of separation and thus different formalisms to be used for information and computation viewpoint specification languages. TINA-C with quasi-GDMO+GRM [18] and ODL [19]; and ITU-SG15 [9] realize such a separation. However, in practice the two issues are often merged in existing formalisms e.g. in GRM + GDMO/ASN.1 and in the SNMP-SMI. In contrast, CORBA-IDL gives only operation signatures, leaving the information model unspecified. Though GDMO allows the specification of information models[2], a given existing GDMO specification may not be complete from the information viewpoint, e.g. there is no guarantee that from an existing GDMO specification all the attributes are there in order to fully specify the behavior of a given action. Therefore, obtaining an information specification may be more complex than just removing from a GDMO specification the computational or any other unwanted aspects.

Towards a Generic and Usable Framework : The approach taken here is to integrate the existing information and interface specification models as-is, and to finalize such specifications according to what would be its projection on the "ideal" ODP perspective. This process may require to separate concerns, to complete unspecified parts.... Integrating GDMO, GRM, IDL as-is brings extraneous work at the beginning to customize the generic model. But on the long run, this provides for a much more usable specification framework. Specifiers/users can write/read specifications close to their universe of discourse. This results as specifications being more easy to write, read and understand than if a general common framework was used with the burden to have to convert, all the time, between concepts of a domain to too general concepts.

Instantiation and validation of the generic model has been done in the context of the Telecommunication Management Network (TMN) [17]. TMN is based on the OSI Systems Management Architecture including GDMO, ASN.1 extended with relationships (GRM) and CMIS/P for communication issues.

Plan of the paper :

1. In section 2 the generic behavior specification template notation is introduced along with its constituent clauses.
2. In section 3 the generic operational semantics is presented, emphasis is put on its visual character because this is a very important factor for the usability of the execution environment.
3. Section 4 presents features about the user environment. The different execution modes are listed. Basic level debugging is presented and hints towards the provision of improved debugging with backtracking are also given.
4. Section 5 describes how the main features of the generic model are realized using *Scheme*.

[2]In fact GDMO is a super-set of what is actually needed for an information specification language. That is the reason why ITU-SG15 have defined the GDIO (Guidelines for the Definition of Information Objects) notation and TINA-C have defined the quasi-GDMO+GRM notation.

2 Generic Specification Framework

The generic behavior specification framework is based on Guarded Behaviors. A **Guard** specifies an acceptance condition for a **Behavior Body** to be executed. Both guard and body are clauses of the proposed behavior template notation. This model provides direct support for nondeterministic behavior specifications because at a given step nothing prevents from several guard conditions to be fulfilled. Bodies define rewrite pieces of code on a Distributed Shared Memory (DSM). No assumption is made on the DSM. It can be organized in any suitable way according to the information model used, e.g. flat variables (SNMP-SMI), Objects (OSI-MIM, CORBA) eventually extended with relationships (GRM [12], OMG Relationship COS [15]). Behaviors specify the reaction of the system to the occurrence of an event, i.e. service request from a client, a failure in an equipment. . . . In the following the term *trigger* and *event* are used interchangeably. An event is represented in behaviors as a message, this is the reason why the notion of *event-message* is also used below. Event messages typically represent operations that can be invoked at computational object interfaces. Event messages can naturally be sent in behavior bodies thanks to well identified primitives. As one would have noticed, analogy is rather obvious with production systems that can be found in rule based expert systems from the AI field or in Event-Condition-Action (ECA) rules from the active database field [10, 6]. However, goals are not the same because in such domains, the production system is used to support actual operation, management, control. . . applications. Here, an executable specification framework is used to perform validation. As argued by Fuchs in [8], a declarative framework which is exactly what procures a production system is a valuable option for specification executability.

Data Oriented Models : DOMs describe the data in a system and how actions exercise their behaviors onto the data. This is a departure from the process oriented model which have been considered as a conventional choice for executable specifications [3]. In DisCo project [13] the advantages of data versus process oriented models are fully justified. This argumentation is valid at least for functional behavior properties : A DOM avoids thread partitioning, and to determine which thread of control is responsible for which part of the data. Here, the control flow of executed action behaviors is completely driven by data dependencies. Thus maximal nondeterminism is implicitly assumed, and if restrictions with respect to the nondeterminism allowed in the system are to be observed, they have to be given explicitly. Thus what is explicit is the additional constraints (synchronization, atomicity, sequentiality. . .) to observe, this is a better approach from the specification point of view because, in some way, it is minimalist and avoids over-specification. When dealing with functional properties of a distributed system, one is not interested in modeling how object and processes are distributed. More exactly, one would be interested to make the less assumption as possible about distribution. The advantage is that all the distribution alternatives remain implicitly allowed, so that they remain available to all be legitimately examined at the right place, i.e. when engineering issues are treated.

In contrast, Process Oriented Models (POM) – e.g. Algebraic specifications (Lotos, CCS), CSP and FSM models (SDL, Estelle) – include engineering issues such as behavior and object partitioning among processes. This is necessary to deal with non-functional behavior properties like timing and real time constraints. Thus DOMs and POMs are not to be opposed, they are in fact complementary techniques. DOMs being more abstract are more likely to be used to treat information and computation viewpoint issues. POMs allowing to include behavior properties dependent on the actual distribution of objects are more adapted to deal with engineering issues.

Making Use of Assertions : Finally, the behavior template is extended with **pre** and **post** condition clauses. They are conditions specified just like guards but intended to a different goal, i.e. to check for the correctness of performed executions.

Recap : a behavior template is defined as a tuple with the following clauses : *beh* ≡ *¡label, when, pre, body, post¿*. The next section defines more in detail its operational semantics, i.e. how behaviors defined according to this generic template are exercised during execution.

3 Generic Visual Operational Semantics

Because the selected approach for the validation of behavior specifications relies on the execution of test suites and on the observation of their outcome, a good test execution environment is of paramount importance. To this end, two complementary views are provided to give the user a complete description of the whole system's state : the **Object View** and the **Execution View**.

3.1 Object View

The execution model is a data oriented model, into which behaviors can be seen as rewriting rules on a globally available distributed shared memory (DSM). Each time something occurs in a behavior, its eventual effects can be reflected on the underlying DSM. According to the way the DSM is structured, e.g. with objects and relationships, it can be visually represented as a graph. Then any change on this graph can be highlighted to the user which naturally lends to animation. Figure 1 gives an example of an Object View snapshot[3]. This example is taken from a TMN case study dealing with service provisioning in a generic network infrastructure. This object and relationship graph is instantiated according to TMN information models : GDMO for Managed Objects (drawn as ovals), and GRM for relationships (drawn as rhombuses) and roles (drawn as rectangles). Parts of the object view are hidden because they have been cut. However, they can be restored as needed.

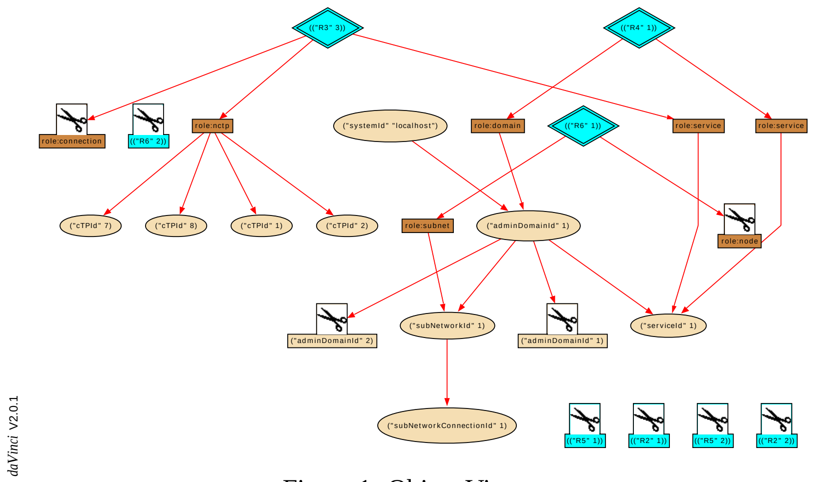

Figure 1: Object View.

3.2 Execution View

This view is defined by the concept of Behavior Execution Tree (BET), which is itself a tree of Behavior Execution Nodes (BEN). A BEN represents a behavior at a given step of the execution in its body code. In this code, two kinds of primitives can be used to :

- invoke operations on a computational interface, this is done by sending event messages.
- interact with the underlying information objects, this is done by using the DSM-API.

Behaviors may have been defined to model the required reaction of the system to the emission such event messages. This new behaviors being executed define new behavior execution nodes that are visually represented as children of the original parent BEN. The BET can be used to provide for a visual description of a complex behavior propagation. It can also be seen as a visual representation of the way the operational semantics of the proposed behavior specification paradigm is exercised. The operational semantics used is based on the interleaving execution model. That is, execution steps of running behaviors can be interleaved in any arbitrary order. Atomic execution steps are defined by behavior body code delimited by interactions with the DSM, i.e. usage of primitives from the DSM-API. Finer grain execution steps are not needed (e.g. at the level of *Scheme* instructions) because it is assumed that behaviors can interfere only on the basis of DSM interactions[4]. Figure 2, gives an example of a behavior execution tree. Each node is documented with a unique identifier accompanied with the label of its corresponding behavior

[3]The graph visualization tool *daVinci* [5] is used to provide the object view. It is also used to provide the execution view (see section 3.2).

[4]Note that this assumption is valid only if access to shared behavior variables is limited to the DSM (whatever the way it is implemented). Non careful access within behavior code to global *Scheme* variables can easily violate this assumption.

and its state : ¡i beh-label state . . . ¿. The different values for the BEN state field are explained below. BEN ids can be interpreted as a birth date. This gives an immediate view about how the BET was developed. To get a visual representation, BENs can be labeled and colored differently according to their state. Additional attributes accessible from each BEN can also be added by the user to customize what is actually displayed for each BEN developed. The BET shown in figure 2, illustrates through a toy example how the computation of the value of the Fibonacci function for the integer 3 is being processed. In addition, appendix B gives :

- the corresponding behaviors.
- the unique event message *fibmsg* used to trigger recursive Fibonacci computations.

Note that, this toy example illustrates only control aspects, that is the reason why there is no use of underlying information object accessed through the DSM-API.

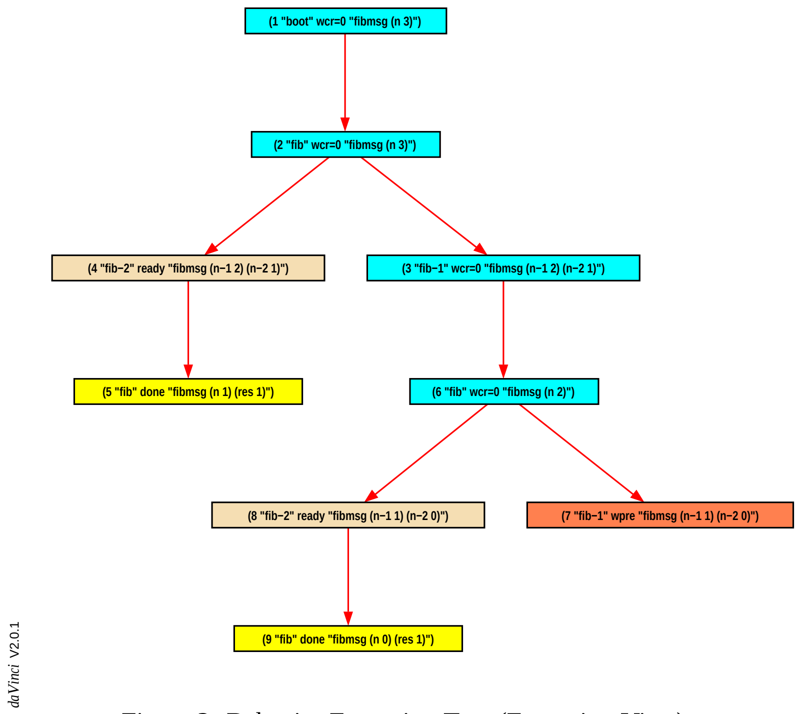
(1 "boot" wcr=0 "fibmsg (n 3)")
(2 "fib" wcr=0 "fibmsg (n 3)")
(4 "fib−2" ready "fibmsg (n−1 2) (n−2 1)")
(3 "fib−1" wcr=0 "fibmsg (n−1 2) (n−2 1)")
(5 "fib" done "fibmsg (n 1) (res 1)")
(6 "fib" wcr=0 "fibmsg (n 2)")
(8 "fib−2" ready "fibmsg (n−1 1) (n−2 0)")
(7 "fib−1" wpre "fibmsg (n−1 1) (n−2 0)")
(9 "fib" done "fibmsg (n 0) (res 1)")
daVinci V2.0.1

Figure 2: Behavior Execution Tree (Execution View).

The different ways into which a BET can be further developed, from a given state, are completely determined by the state of its constituents BENs and their possible individual evolutions. Thus, at this point it is worth to describe more precisely a BEN and its transition diagram. BEN properties, are listed in figure 3.

record	Fields
BEN	¡*id, state, beh, children, parent, step, wcr=0, bec*¿
STEP	¡*beh-cont, beh-src-line*¿
BEC	¡*evt-msg, dsm-context*¿
BEH	¡*label, when, pre, body, post*¿

Figure 3: Behavior Execution Node Properties.

3.3 BEN Fields

the state field defines the operations that are possible at a given step, i.e. that the user can invoke on the execution node. Its value is one of the list *(ready, wcr=0, wpre, wpost, done)*. A BEN in the *ready* state means that it is ready to execute the next step in its body code. All BENs in this state form the *ready-list* of BENs ready to execute. When the *ready-list* becomes empty, the behavior execution has completed. A terminating state has been reached. Transitions between BEN state values can be described as a finite state machine (see figure 4).

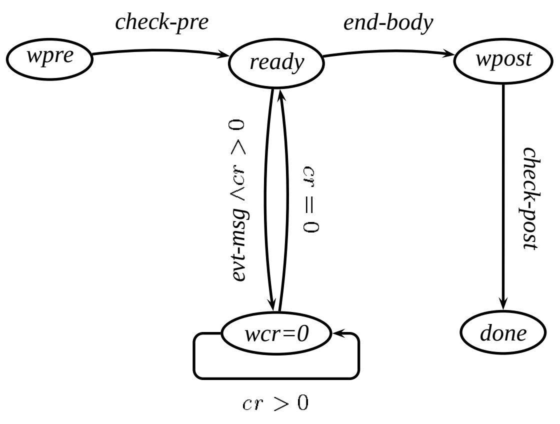

Figure 4: Behavior Execution Node FSM.

the step field represents the current stage of processing in the behavior body code. It is composed of :

- *beh-src-line* field references to the line number of a behavior source file, thus enabling source level debugging.
- *beh-cont* field is a *Scheme* continuation (see section 5) that is called to resume the execution of this behavior in order to execute its consecutive steps.

the cr field is a decreasing counter. It gives for a BEN that has sent an event message to be propagated, the *child residue*, i.e. the number of child BENs currently being executed and that the (parent) BEN is waiting for completion. As shown on the BEN FSM in figure 4, while $cr > 0$ the parent BEN is not allowed to resume the execution of its body code.

the beh field is the behavior itself that was fetched. It is itself a record with fields values representing internally what has been defined in the behavior clauses using the generic behavior template presented in section 2.

the Behavior Execution Context field (BEC) is a completely opaque data for the generic kernel that is given as argument to each evaluation of behavior clauses. This concerns the guard (*when* clause), *pre*, *body* and *post*. BEC semantics are defined according to the application domain into which one intends to use the generic kernel, e.g. the information and computation model used. From our experience in the TMN context, it is typically composed of :

- *evt-msg* is the message sent that caused this behavior to be fetched. It is, of course, dependent on the services invocation primitives supported by the underlying computational model. So values for this field are out of the scope of the generic model.
- *dsm-context* represents the execution scope of this behavior on the DSM. This field is dependent on the DSM or information model used. This typically references a variable, an object, a group of objects, a relationship or any other data abstraction of interest for a given information model.
- . . . any other property required to customize the generic model.

4 Execution Modes and Debugging

The object and execution views are the basis for a powerful debugging tool. The object view enables the user to inspect that the DSM components are assigned with correct values at any execution step. The execution view gives a proper representation of the nondeterminism occurring, i.e. concurrency, choice or unordering along performed executions. Concretely, the user can get a direct feeling of the nondeterminism by seeing all the BENs in the current *ready-list* that are highlighted on the BET. At each step, the user has a full control about which execution branch of the BET should be explored. Other basic debugging features such as trace and breakpoint facilities are also possible.

This enables the following execution modes to be provided :

- The **user driven** mode enables the user to develop the BET as wished. At each step, the user selects the BEN (which belongs to the ready-list) from which its next step is to be executed. In this mode, it is also possible to terminate the execution of a BEN according to one of the policies described below. This is a required feature in order to speed up the development of parts of the BET, e.g. because they are already debugged. . . .
- The **random walk** policy enables to develop the BET randomly, and reach one of the possible terminating states.
- The **fixed walk** policy develops the BET according with a fixed strategy, e.g. depth or breath first. This is useful for early stages of debugging, to force the execution to follow always the same path. This avoids to be annoyed by the other problems caused by nondeterminism that one would prefer to fight at a later stage.

Improved Debugging Support : Basic source level debugging support do not prevent from wasting precious debugging time in setting break-points in backward order and re-running the program, or in stepping over the whole execution flow, until the erroneous code is reached. In contrast, an execution backtracking facility in interactive source level debuggers allows users to mirror their though processes while debugging [2]. This enables to work backwards from the location where an error is manifested and determine the conditions under which the error occurred. Such a facility also allows a user to change program characteristics and re-execute from arbitrary points within the program under examination (a ”what-if” capability). This very powerful dynamic debugging facility requires two major underlying mechanisms : **control backtracking** and **data backtracking**. Here are given some hints on how such mechanisms could be incorporated to the generic meta-model :

- **Control backtracking** is easy to provide using *Scheme* continuations (see section 5). It is just needed to store the history of steps in each BEN, rather than only the current one. Because each step contains a continuation corresponding to each execution stage in the behavior body, it is possible to go back to any previous execution stage by calling back the corresponding continuation.
- **Data backtracking** is another issue. This requires to be able to restore the DSM states of each behavior execution step. Such a mechanism is totally DSM implementation dependent. If the DSM is implemented on top of database, data backtracking could probably be implemented in terms of transaction processing[5]. If the DSM is directly implemented in the *Scheme* environment, another approach to implement data backtracking is to provide for a data undoing function. However, in order to avoid to re-implement this mechanism for each DSM model, an idea would be provide a generic facility on top of which any form of DSM could be realized. This would enable to define data undoing on the generic storage facility. A notion of generic repository of information based on hash-tables is already being used for the currently used TMN-based instantiation of the generic model. Though not yet implemented, data undoing function should not be too difficult to implement on such generic repositories.
- An execution trace is also needed besides the BET. The backtracking process has to follow this trace in backward direction. This trace can be simply implemented as a stack of BEN identifiers, from which execution steps can be retrieved in backward order.

5 Realization in *Scheme*

Why *Scheme* ? It is a simple, clear and sound programming language. Its syntax is reduced to the very necessary and it should not be long to learn (the standard [4] is less that 50 pages). *Scheme* provides the required programming support for behavior coding (control structures, variable notations. . . that does not have to be reinvented). In addition, if the *Scheme* Library is used [7], facilities such as quantifiers are available. This can be used to simplify the specification of logical expressions (guards, pres and posts).

[5]In effect, a database system offering only a transaction concept is sufficient : Marking a data state is done by starting a new transaction and using a counter for the number of opened transactions. Backtracking is done by aborting a sufficient number of transactions. This works, but in some way the transaction mechanism is abused, since transactions are never committed as long as the behavior propagation is processed.

The Macro System : has been used to define the behavior specification template presented in section 2. *Scheme* provides high level macros that enables to define expressive language extensions and how they are to be expanded to pure *Scheme* code.

Closures or λ-expressions : provide a nice abstraction to store the different pieces of executable specification code present in guards, bodies, pres and posts. All this clauses are expanded and stored as closures (of one argument) in the behavior repository. At evaluation, such closures are called with the current behavior execution context (BEC) as argument. The BEC contains all the relevant information required for the evaluation of such closures.

Behavior Instrumentation : To enable source level debugging behavior instrumentation have to be performed just before behaviors are loaded in the behavior repository of the execution environment. This provides during execution references to the original behavior source code.

***Scheme* Continuations :** Though small and simple, *Scheme* provides in the language itself a powerful concept (continuations) and an associated construct (`call-with-current-continuation` often abbreviated as `call/cc`) that enables to set up and store any execution context and the way to go back to this context, by simply calling it as if it was a usual function. Continuations are used to provide the basic control mechanisms needed in order to implement basic debugging support (i.e. stepping and breakpoints). In addition, continuations are extremely useful for implementing a wide variety of advanced control structures, such as complex forms of backtracking. Therefore, they can also be used in the realization of the improved debugging with backtracking support to provide for a straightforward implementation of the control backtracking part. Appendix A gives a quick overview through a simple example on the way continuations are working.

6 Conclusion and Further Issues

The generic behavior specification and validation environment defines a working and usable generic model. It is designed with ODP principles in mind, to allow for a clean distinction between information and computation issues in behavior modeling. The proposed behavior template is based on guarded action behaviors. This is a quite classical approach, and similar constructs have been used for general production systems such as rule based expert systems or ECA-rules in active database systems. Such a declarative framework is also a natural option for executable specifications [8]. In addition, in order to perform validation, the behavior model is enriched with assertions that can be checked along behavior execution paths. This also follows the approach advocated by ITU-SG15 work [9]. This generic behavior model has been first instantiated in the TMN context (GDMO/ASN.1, GRM and CMIS/P) and used in several TMN-based case studies. Work is being done to customize the model towards CORBA based systems. This should prove again more effectively the genericity of the approach. This concerns to reuse and check for the applicability of not only, the specification framework but also, the implemented operational semantics and execution environment.

A further issue is to effectively implement the improved debugging with backtracking facility. This may reveal as a very interesting mechanism to base the support for a powerful problem explanation tool. Though the proposed behavior model can be very useful in order to deliver better distributed behavior specifications (which is in itself a real win), one would contest that the approach is not so formal. In effect, providing another specification language, even equipped with an operational semantics and execution environment, is far from being a sufficient condition towards formality. Without the ability to do some kind of formal reasoning such as analytical proof or exhaustive search, there is little place for actual formality. The executable framework is naturally more oriented towards an exhaustive search approach. In particular, the backtracking facilities can be used to do exhaustive behavior analysis of all behavior execution paths that can be followed for a given test case. This would enable one to highlight all the cases where assertions are violated. And, even if no assertion is violated, it may be interesting to examine all the terminating configurations of the system. May be some of them exhibit unwanted behavior executions. But as usual when exhaustive search is performed in the context of an interleaving execution model, the classical state explosion problem occurs. It is worth to see how well known techniques, e.g. partial order methods [11] could be applied to the generic model considered.

Acknowledgments : The author is indebted to the three anonymous reviewers for their helpful comments and suggestions.

References

[1] A CASE Environment for TINA-oriented Applications, 1996. Available at http://andromeda.cselt.stet.it:8080/ace/ACE.html.

[2] Hiralal Agrawal, Richard A. DeMillo, and Eugene Spafford. An Execution Backtracking Approach to Program Debugging. Technical Report SERC-TR-22-P, Software Engineering Research Center Purdue University, September 1990.

[3] J. P. Briand, M. C. Fehri, L. Logrippo, and A. Obaid. Executing LOTOS Specifications. *Protocol Specification, Testing and Verification VI, IFIP 1987*, pages 73–84, 1987.

[4] W. Clinger and J. Rees. $Revised^4$ Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), 1991. Available at http://www.cs.indiana.edu/scheme-repository/doc/standards/r4rs.ps.gz.

[5] The Interactive Graph Visualization System daVinci. Available at http://www.informatik.uni-bremen.de/˜inform/forschung/daVinci/daVinci.html.

[6] Klaus R. Dittrich, Stella Gatziu, and Andreas Geppert. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. Technical report, University of Zurich, Dept. of Computer Science, 1995. Available at http://www.ifi.unizh.ch/techreports.

[7] T.R. Eigenschink, D. Love, and A. Jaffer. SLIB: The Portable Scheme Library, 1994.

[8] Norbert E. Fuchs. Specifications are (preferably) executable. Technical Report 92, University of Zurich (CS Dept.), 1992. Available at ftp://ftp.ifi.unizh.ch/pub/techreports/.

[9] Management of the Transport Network – Application of the ODP Framework, ITU-T G851-01, 1996.

[10] Andreas Geppert, Stella Gatziu, Klaus R. Dittrich, Hans Fritschi, and Anca Vaduva. Architecture and Implementation of the Active Object-Oriented Database Management System SAMOS. Technical report, University of Zurich, Dept. of Computer Science, 1995. Available at http://www.ifi.unizh.ch/techreports.

[11] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems* – An Approach to the State Exploision Problem. PhD thesis, Université de Liège, Faculté des Sciences Appliquées, 1995. Available at http://www.montefiore.ulg.ac.be/services/verif/papers/thesis.ps.Z.

[12] ISO/IEC JTC 1/SC 21, ITU X.725 – Information Technology – Open System Interconnection – Data Management and Open Distributed Processing – Structure of Management Information – Part 7 : General Relationship Model.

[13] Hannu-Matti Jarvinen and Reino Kurki-Suonio. DisCo Specification Language: Marriage of Action and Objects. In *Proc. of 11th International Conference on Distributed Computing Systems*, Arlington, Texas, may 1991. IEEE Computer Society Press. Available at http://www.cs.tut.fi/laitos/DisCo/DisCo-english.fm.html.

[14] Object Management Group : Object Analysis and Design RFI, 1995. OMG TC Document 95-9-35.

[15] OMG Relationship Service Submission, 1994. Revised submission to the Object Services Task Force RFP2 by Bull, Hewlett-Packard, Olivetti, IBM, SNI, and SunSoft. OMG Document 94-05-05, available at http://www.omg.org/docs/1994/94-05-05.ps.

[16] Basic Reference Model of ODP – Part 1: Overview and Guide to Use of the Reference Model, ISO 10746-1, ITU X.901.

[17] Dominique Sidou, Sandro Mazziotta, and Rolf Eberhardt. TIMS : a TMN-based Information Model Simulator, Principles and Application to a Simple Case Study. In *Sixth International Workshop on Distributed Systems : Operations & Management*, Ottawa - Canada, 1995. IFIP / IEEE. Available at http://www.eurecom.fr/˜tims/papers/dsom95-paper.ps.gz.

[18] TINA Information Modelling Concepts, 1994. Available at http://www.tinac.com.

[19] TINA Object Definition Language (TINA-ODL) Manual, 1995. Available at http://www.tinac.com.

[20] User Manual for the IFAD VDM-SL Toolbox, 1996. Available at http://www.ifad.dk/products/toolbox.html.

A Continuations, a Simple Example

The procedure `set-cont-here` sets up and returns a continuation at the following statement from which it is called. As shown below, `c1` can be re-called back to redo the three `displays`, `c2` for the two last ones and `c3` for the last one.

```
scm-prompt> (define (set-cont-here)
               (call-with-current-continuation
                 (lambda (c) c)))
#<unspecified>

scm-prompt> (begin ; this executes the "displays" and sets up the continuations.
              (newline)
              (define c1 (set-cont-here))
              (display "Hi 1!!!") (newline)
              (define c2 (set-cont-here))
              (display "Hi 2!!!") (newline)
              (define c3 (set-cont-here))
              (display "Hi 3!!!") (newline))
Hi 1!!!
Hi 2!!!
Hi 3!!!

;;; continuations are Scheme objects, whose value can be printed.
;;; This is done here by simply making a list of their values.
;;;
scm-prompt> (list c1 c2 c3)
(#<cont 316 @ 7d390> #<cont 316 @ 77b38> #<cont 316 @ 78070>)

;;; calling back continuation c1
;;;
scm-prompt> (c1 c1)
Hi 1!!!
Hi 2!!!
Hi 3!!!

;;; calling back continuation c2
;;;
scm-prompt> (c2 c2)
Hi 2!!!
Hi 3!!!

;;; calling back continuation c3
;;;
scm-prompt> (c3 c3)
Hi 3!!!

scm-prompt> (quit) ; quitting the scheme interpreter!
```

B Fibonacci Toy Example : Event Messages and Behaviors

```
;;; unique event message used, it contains both n, n-1 and n-2 fields,
;;; and the result.
;;; Behaviors use only the field(s) they are interested in.
;;;
(genrec:define fibmsg n n-1 n-2 res) ; this defines the fibmsg record.

;;; fib behavior.
;;;
(define-behavior "fib"
  (when (and (fibmsg:isa? (msg)) (specified? (fibmsg:n (msg)))))
  (pre)
  (body (cond ((< (fibmsg:n (msg)) 2) (fibmsg:res! (msg) 1))
              (else (let ((becs (msg-send (fibmsg:make2 ‘(n-1 ,(- (fibmsg:n (msg)) 1))
                                                        ‘(n-2 ,(- (fibmsg:n (msg)) 2))))))
                      ;; collection of result for fibmsg:n from returned
                      ;; fibmsg:n-1 and fibmsg:n-2
                      (fibmsg:res! (msg) (+ (val:get becs ’(0 msg res))
                                            (val:get becs ’(1 msg res))))))))
  (post))

;;; fib-1 behavior, treats the computation of (fib n-1).
;;;
(define-behavior "fib-1"
  (when (and (fibmsg:isa? (msg)) (specified? (fibmsg:n-1 (msg)))))
  (pre)
  (body (let ((becs (msg-send (fibmsg:make2 ‘(n ,(fibmsg:n-1 (msg)))))))
          (fibmsg:res! (msg) (val:get becs ’(0 msg res)))))
  (post))

;;; fib-2 behavior, treats the computation of (fib n-2).
;;;
(define-behavior "fib-2"
  (when (and (fibmsg:isa? (msg)) (specified? (fibmsg:n-2 (msg)))))
  (pre)
  (body (let ((becs (msg-send (fibmsg:make2 ‘(n ,(fibmsg:n-2 (msg)))))))
          (fibmsg:res! (msg) (val:get becs ’(0 msg res)))))
  (post))

;;; function to run the process until some intermediate computation state.
;;;
(define (fib:run-example)
  ;; sending to the propagation engine the computation of (Fibonacci 3)
  ;;
  (msg-send (fibmsg:make2 ‘(n 3)))
  (ben:next 2) (ben:next 2) ; stepping on the BENs.
  (ben:next 4) (ben:next 4)
  (ben:next 5) (ben:next 5) (ben:next 5)
  (ben:next 3) (ben:next 3)
  (ben:next 6) (ben:next 6)
  (ben:next 8) (ben:next 8)
  (ben:next 9) (ben:next 9) (ben:next 9)
  (bet:view)) ; viewing the BET.

;;; calling fib:run-example.
;;;
(fib:run-example)
```