Precise Semantics for a Behavior Model in the Context of Object Based Distributed Systems.

D. Sidou (Institut Eurécom, sidou@eurecom.fr)

September 2, 1997

Contents

1 Introduction

The objective of this position paper is to define a precise semantics for a *behavior language* (BL) to be used in the context of *object based distributed systems* (OBDS). This behavior language was defined and used in the context of the the TIMS project[1][Eberhardt et al.97]. OBDS considered are TMN systems and CORBA systems. The modeling approach follows the work done in ITU-T SG4 [G851 0196]. The modeling is limited to functional behavior properties, no timing and real time issues are considered. The approach towards validation is based on executable specifications, onto which test cases can be submitted and run thanks to an execution environment. The specification framework is declarative in the sense that *actions*[2] are specified one by one in an independent way using a form of event-condition-action ECA-rule. Such a behavior model has been qualified as data oriented [Jarvinen et al.91] in opposition to process oriented models often used in executable specifications frameworks (e.g. Lotos, SDL). The reason is that ECA-rules completely abstract away any notion of process or thread of execution. In effect an ECA-rule simply tells that when a given event occurs and the data is in a proper state then the action is executed, which may in its turn send some other event or update some data. The association between processes or threads of control and rules is left totally unspecified. It turns out that such a model is particularly adapted

[1] This work was done in the context of the TIMS project. TIMS stands for TMN-based Information Model Simulator. This project is a collaboration between Eurécom Institute and Swiss Telecom PTT. It is supported by Swiss Telecom PTT.

[2] Action is considered in the ODP [Rm odp2] sense, i.e. as something which occurs, involving one or more objects, and exercising itself at any level of granularity (atomic or not).

to the modeling of functional issues, e.g. state transitions on the information model. For functional aspects it is not important to know the actual distribution of objects among threads and threads among locations .... Indeed, it may reveal tedious to include such aspects, and this may even lead to overspecify by adding irrelevant non-functional constraints.
In the context of an executable specification framework, a **precise execution semantics** is essential. The reason is that even for relatively small behavior specifications, behavior executions can be complex and unpredictable. In effect, an event can trigger many reactions. Unless otherwise specified, the default execution model interleaves the execution of actions. In addition, each reaction can send in its turn new events, that may trigger other reactions. Finally, the execution model leads to complex behavior propagations, just as what can be observed in a real distributed environment. In our model this semantics is defined by an algorithm called the behavior propagation engine (BPE) algorithm, it works like a forward search inference engine, i.e. it performs behavior execution steps until saturation, i.e. nothing remains to be done.
The ECA-rule model is a result mostly coming from the *active database management systems* (ADBMS) community. In this context it is natural to reuse the results also in terms of the rule processing semantics that have emerged in this community. Survey papers about rule processing semantics in the context of ADBMSs are [Hanson et al.92, Dittrich et al.95]. An interesting example of ADBMS is the SAMOS system [Geppert et al.95].

2 Behavior notation

Here is the notation used in our behavior model :

$$\begin{aligned} define-behavior \Rightarrow (&\texttt{define-behavior}\ label-spec \\ &scope \\ &when \\ &exec-rules \\ &pre \\ &body \\ &post) \end{aligned}$$

The **`scope, when`** clauses correspond to the event and condition of the ECA-rule model, but packaged differently. The **`body`** is the ECA-rule action. In addition, our behavior model allows to specify safety properties by means of assertions (i.e. pre and post conditions) that are checked before and after the behavior body is executed during simulations.

3 Triggering Event Messages and Reaction Semantics

A behavior in our model can always be reduced to some form of reaction to a triggering event that is represented by a message. That is the reason why the term *triggering event message* (TEM) has to be introduced. The different forms of reactions to such TEMs, called reaction semantics, are then described.

3.1 Triggering Event Messages

A TEM is sent to the system either from its environment or from within a behavior. Different levels of event messages can be distinguished. In particular in the context of OBDSs, it is worth to distinguish between TEMs sent to computational objects (i.e. potential units of distribution in ODP, with well defined interfaces), and between TEMs sent to information object (i.e. universe of discourse in ODP, e.g. relationships, roles or simple information objects with state). It is interesting to specify most of the behavior at the information viewpoint, and to reuse such specification at the computational viewpoint level. A typical illustration of this approach is [G851 0196], where at the computational viewpoint level the proposed semi-formal behavior notation template makes explicit references to dynamic schemas specified at the information viewpoint level.

3.2 Reaction Semantics

Two important issues used to characterize a reaction semantics are the *fetching phase* and the *coupling mode*. These issues are specified using the **`exec-rules`** clause of the proposed behavior notation template. Note that, similar concepts can be retrieved in the ADBMS literature [Hanson et al.92]. The fetching phase is called the event-condition (EC) coupling mode and our concept of coupling mode is called the condition-action (CA) coupling mode. But before these issues can be described, it is first required to introduce a specific kind of reaction semantics : the **`is-trigger`** reaction semantics.

3.3 The `is-trigger` Reaction Semantics

Because there is no builtin semantics pre-defined for any event message[3] entering the system, a first type of reaction semantics is used to define what has to be actually performed when a given event message is sent to the system. Very often this consists to do very simple and basic things. For instance, the execution semantics of a `ivpmsg-set` message consists to make some change in the data store of the system for a given object and attribute to a given value. This reaction semantics is called the **`is-trigger`** semantics just because it defines the semantics that is directly associated to the execution of the triggering event message.

3.4 Fetching Phase

The fetching phase specifies when behaviors are candidate for fetching with respect to the occurrence of the triggering event message. Fetching can be specified to occur immediately (**`phase-i`**), in that case fetching is done as soon as the trigger is sent to the system. Otherwise, fetching can be deferred (**`phase-ii`**). One kind of deferred fetching[4] has been defined. In our behavior model, a behavior whose fetching phase is deferred is candidate for fetching only when all the behaviors executing with the **`is-trigger`** reaction semantics have completed. Note that completion means that the execution of the entire behavior block – i.e. **`pre`**, **`body`** and **`post`** – has terminated.

3.5 Coupling Mode

Once a behavior has been fetched, another important issue is to determine how its behavior block has to be executed. Here two types of reaction semantics can be distinguished :

1. The **`coupled`** reaction semantics specifies that the execution of the behavior block is coupled with the caller, i.e. the behavior that has sent the trigger. An important consequence is that the calling behavior block execution is blocked until all the coupled reactions terminate. Another important consequence is that this results as a cascaded behavior execution model, as soon as new event messages are sent from a behavior body.

2. The **`uncoupled`** reaction semantics specifies that the execution of the reaction is to be done in an independent thread of execution of the calling thread. Being able to model such reaction semantics is very useful in a distributed environment, because distribution lends naturally to independent threads of execution.

To completely define a coupled reaction, one has to specify when the execution of the behavior block is expected to initiate and terminate. For an uncoupled reaction, only the initiation issue is relevant. In both cases such parameters are specified relatively to the **`is-trigger`** reaction semantics. In effect, the execution phase during which behaviors with the **`is-trigger`** reaction semantics are executing is called the **`during-trigger`** phase. As shown in figure 1, the time during which the reactions to a trigger are being executed can be partitioned into three phases : the **`before-trigger`**, the **`during-trigger`**, and the **`after-trigger`** phases.

[3] That means that the default reaction to an event message sent to the system is to do nothing.
[4] One could imagine other ways to do deferred fetching, however it is not clear whether they are actually useful or not.

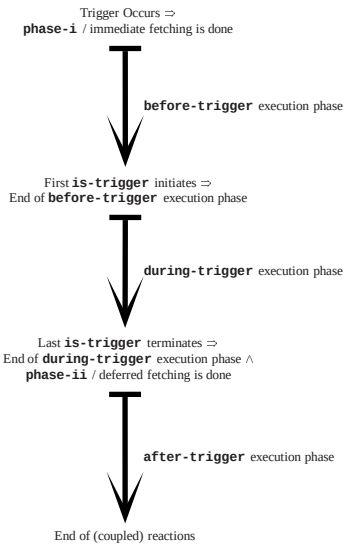

Figure 1: Trigger Reaction Phases

3.6 The `exec-rules` Clause

In conclusion, the different reaction semantics are defined using two parameters : the fetching phase and the coupling mode. As indicated before, they are specified in the **`exec-rules`** clause as follows[5]:

$$\begin{aligned} exec-rules \Rightarrow\ & (\textbf{exec-rules}\ fetch-phase\ coupled)\ \mid \\ & (\textbf{exec-rules}\ fetch-phase\ uncoupled) \end{aligned}$$

$$\begin{aligned} fetch-phase \Rightarrow\ & (\textbf{fetch-phase}\ i)\ \mid \\ & (\textbf{fetch-phase}\ ii) \end{aligned}$$

$$\begin{aligned} coupled \Rightarrow\ & (\textbf{coupled}\ before-trigger)\ \mid \\ & (\textbf{coupled}\ before-trigger\ during-trigger)\ \mid \\ & (\textbf{coupled}\ before-trigger\ after-trigger)\ \mid \\ & (\textbf{coupled}\ during-trigger)\ \mid \\ & (\textbf{coupled}\ is-trigger)\ \mid \\ & (\textbf{coupled}\ during-trigger\ after-trigger)\ \mid \\ & (\textbf{coupled}\ after-trigger) \end{aligned}$$

$$\begin{aligned} uncoupled \Rightarrow\ & (\textbf{uncoupled}\ before-trigger)\ \mid \\ & (\textbf{uncoupled}\ during-trigger)\ \mid \\ & (\textbf{uncoupled}\ after-trigger) \end{aligned}$$

[5] In coupled mode if only one execution phase is given, that means that the behavior block is intended to initiate and terminate in that phase. Note also that **`(coupled is-trigger)`** defines a behavior with the **`is-trigger`** execution semantics.

4 Behavior Propagation Engine (BPE)

In this section the concepts previously introduced are integrated altogether. This results as a complete and precise semantics for the proposed behavior notation. First of all, the BPE is a forward search inference engine, i.e. it performs behavior execution steps until saturation, i.e. nothing remains to be done. This allows to define the concept of behavior execution (also called behavior propagation or simply walk) as follows :

A walk or behavior execution or behavior propagation is a sequence of execution steps performed from the initial state to a state reached at saturation. Let S be the configuration of the system or its state at each step. Let $enabled\_steps(S)$ be the function giving the set of enabled steps that are ready to be executed from state S. The overall algorithm of a forward search inference engine is :
bpe:walk(S)
While enabled_steps(S) $\neq \emptyset$ Do
    S := execute_step(s $\in$ enabled_steps(S), S);
Od

4.1 Towards an LTS semantics :

Now we have to define more precisely what represent and how are represented the important notions of execution step and system's configuration. Once this is done, since system's configurations and execution steps correspond respectively to the states and transitions of a labeled transition system (LTS). This makes the semantics associated to our behavior model being defined operationally by this LTS.

4.1.1 Transitions / Execution Steps

Transitions / execution steps are defined by the atomic pieces of execution. Basically, two kind of execution steps have been identified :

1. assertion evaluation step is a `pre` or a `post` clause that is evaluated.
2. behavior body evaluation steps correspond to the algorithmic pieces of behavior body code delimited by statements sending event messages. A point in the execution of a behavior body where event messages are sent is called a *blocking point*. When such a statement is executed, the execution of the behavior body is blocked until all the coupled reactions to the event messages sent are terminated. A point where all such coupled reactions terminate is called a *resume point*. So, in the end evaluation steps in the execution of behavior bodies are either :
   (a) the execution from its beginning to the first blocking point.
   (b) an execution from a resume point to the next blocking point. It is important to note that this includes the evaluation of the behavior fetching corresponding to the triggering event messages sent at the blocking point.
   (c) the execution from the last resume point to the end of the behavior body.

4.1.2 States / Configurations of the System

The state or configuration of a system is usually partitioned into a *data state* and a *control state*. Here it is important to note that the data state can be abstracted away. The reason is that the BPE machinery is totally independent on any underlying model used to represent the data state and its associated transitions. Indeed the data state can be represented / implemented by any means appropriate, e.g. a database, a repository in main memory .... Therefore in this section, only control states issues are considered, i.e. the structures used to represent the control state and the transitions allowed between such structures.
The important structure used to represent the control state is based on the concept of *behavior execution node* (BEN). A behavior execution node encapsulates a behavior being executed along with the associated information needed at execution time. Then the concept of *behavior execution tree* follows naturally from the fact that each time a behavior sends one or several triggering event messages during the execution of its body code, to the reactions of the system are associated new behaviors that are typically being fetched

for further execution. To each newly fetched behavior is associated in its turn a BEN that is identified as as a child of the original BEN (the message(s) sender BEN). This reflects the causality relationship existing between behavior executions. As a consequence, a natural representation for behavior propagations is a graph. More precisely it is a tree, called the *behavior execution tree* (BET).
The actual structure used to represent a BEN can be described using a record with the following fields :

$$ben \Rightarrow \ll id, children, parent, behavior, bec, state, ccrc, tecs, parent-tec-id, cont, \\ bbody-src-lno, bbody-conts-stack, undo-info-stack, bbody-src-lnos-stack, ops-seqs \gg$$

In the BEN record it is worth to describe the following fields :

1. The `behavior` field points to a record where all the behavior features are stored. This field points to the internal representation of a behavior. This representation is based on a language mapping of the behavior notation defined in section 2 into the *Scheme* [Clinger et al.91] programming language.

2. The `bec` field is the scoping behavior execution context. It was determined when this behavior was fetched. It references all the data needed by the **`when`**, **`pre`**, **`body`** and **`post`** clauses when executed.

3. The `state` field indicates the execution state of the behavior. The states and transitions defined are described in figure 2. Underlined node labels indicate states from which an execution step is possible directly from the BEN. Transition from non-underlined nodes result from transition in related BENs, i.e. child or brother BEN. A particular state is the $wait-ccrc=0$ state, in this state the BEN is blocked because its behavior body has sent event messages and at least one coupled reaction has been fetched. When the `state` field has this value, the BEN state is completed by the fields `ccrc` and `tecs` described below.

4. The *coupled children remaining counter* `ccrc` field is the counter for the number of coupled children whose behavior execution (behavior block) is not yet completed.

5. The *triggers execution controls* (`tecs`) field is a vector. Each element of this vector is called a *trigger execution control* (TEC) record. In such a record is stored all the information used to control the execution of all the behaviors associated to one triggering event message sent. So the number of elements of the `tecs` vector is the number of event messages sent by the BEN behavior body at its last blocking point.

6. The `parent-tec-id` field is the index in the parent `tecs` field vector of the TEC controlling the execution of the behavior in this BEN.

7. The `cont` field is a continuation. A continuation is a powerful control structure that can be used to capture a state of processing, stop it and resume it later on. Continuations are particularly adapted to the control of the execution of behavior bodies. Note that the `cont` is meaningful in the BEN as soon as the BEN is in the *ready* state, because when the BEN enters this state, the execution of its behavior body is ready to resume. A behavior body continuation is captured (and stored in this field) as soon as message sending occurs (blocking point). Because it is a continuation, this field provides the mean to perform the resume[6].

8. The other fields are not useful to describe the operational semantics. They are used for the execution environment, e.g. for debugging.

4.1.3 The Trigger Execution Control

The *trigger execution control* (TEC) is used to schedule the reactions that have been fetched for a trigger. Such schedule consists to go through the different trigger reaction phases that have been identified in figure 1. It is a record with the following fields :

$$tec \Rightarrow \ll children, ccrc, itccrc, phase \gg$$

[6] Since the execution environment used is based on *Scheme* and that continuations are a feature of *Scheme* , it was natural and easy to implement the blocking and resume points using continuations. In other execution environments without continuations it is probably possible to work with other facilities, e.g. threads.

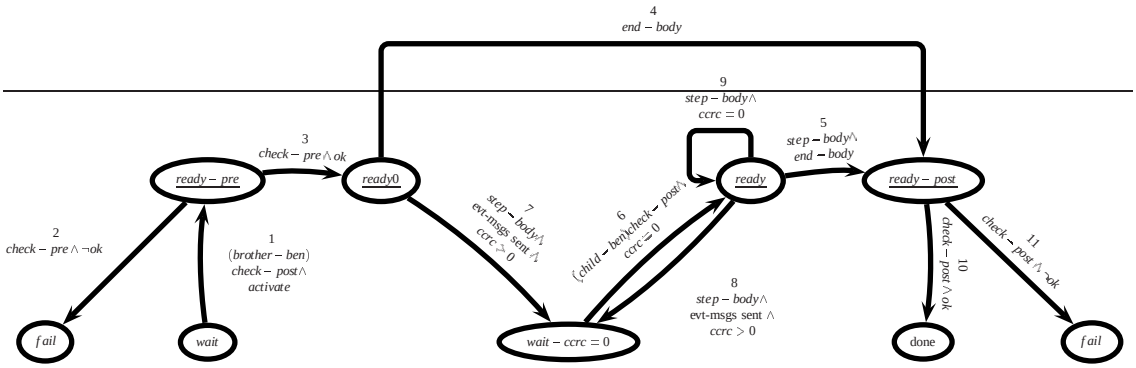

Figure 2: BEN FSM.

In the TEC record it is worth to describe the following fields :

1. The `phase` field is the trigger execution phase as described in figure 1. It is either *before-trigger*, *during-trigger*, *after-trigger*.
2. The `ccrc` field in the TEC record plays the same role as the `ccrc` field in the BEN record, but it counts the number of coupled children for this trigger an in the current trigger reaction phase. When this counter is decremented to zero, phase switching occurs as described in figure 1.
3. The *is-trigger coupled children running counter* (`itccrc`) field is used in the *during-trigger* phase. It counts the number of children still running an is-trigger reaction semantics. This counter is necessary, because when it is decremented to zero, **`phase-ii`** behavior fetching has to occur.

4.1.4 Transition Functions

In summary, to implement the transitions three transition functions have been implemented :

1. The *check – pre* function simply evaluates the pre-condition and stops the behavior propagation in case of failure.
2. The *step – body* function executes the portion of behavior body by resuming the BEN continuation. When the next blocking point is encountered, a new continuation is created to capture a new blocking state, behavior fetching (**`phase-i`**) occurs and the newly fetched BENs are registered as children of the BEN from which the original *step – body* was executed.
3. The *check – post* transition function is probably the more complex, because in this function side-effects on the parent BEN and brother BEN are done to implement (i) trigger execution phase switching, (ii) **`phase-ii`** behavior fetching, and (iii) resume parent BEN execution. Last but not least, this function also evaluates the post-condition and stops the behavior propagation in case of failure.

5 Conclusion

Our work has shown that concepts about rule execution semantics that were established in the context of active database management systems can be reused in the context of our behavior model. Though our goals are oriented towards specification and validation, the experiences resulting from the realization of several case studies proved that such concepts are still reusable in our context. In the end a satisfactory and comprehensive semantics for our behavior model has been obtained.

References

[Clinger et al.91] Clinger (W.) et Rees (J.). – *Revised*^{4} Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, vol. 4 (3), 1991. – Available at http://www.cs.indiana.edu/scheme-repository/doc/standards/r4rs.ps.gz.

[Dittrich et al.95] Dittrich (Klaus R.), Gatziu (Stella) et Geppert (Andreas). – *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.

[Eberhardt et al.97] Eberhardt (Rolf), Mazziotta (Sandro) et Sidou (Dominique). – Design and testing of information models in a virtual environment. *In : The Fifth IFIP/IEEE International Symposium on Integrated Network Management "Integrated Management in a Virtual World"*. – San Diego, CA, USA, may 1997. available at http://www.eurecom.fr/~tims/papers/im97.ps.gz.

[G851 0196] Management of the Transport Network – Application of the ODP Framework, ITU-T G851-01, 1996.

[Geppert et al.95] Geppert (Andreas), Gatziu (Stella), Dittrich (Klaus R.), Fritschi (Hans) et Vaduva (Anca). – *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.

[Hanson et al.92] Hanson (Eric N.) et Widom (Jennifer). – *An Overview of Production Rules in Database Systems*. – Technical report, University of Florida (CIS), 1992. Available at ftp://ftp.cis.ufl.edu/cis/tech-reports/tr92/tr92-031.ps.

[Jarvinen et al.91] Jarvinen (Hannu-Matti) et Kurki-Suonio (Reino). – DisCo Specification Language: Marriage of Action and Objects. *In : Proc. of 11th International Conference on Distributed Computing Systems*. – IEEE Computer Society Press. Available at http://www.cs.tut.fi/laitos/DisCo/DisCo-english.fm.html.

[Rm odp2] Basic Reference Model of ODP – Part 2: Foundations, ISO 10746-2, ITU X.902.