Behavior Simulation and Analysis Techniques for Management Action Policies (or Hints towards the Integartion of Management Action Policies within TIMS).

Dominique Sidou

EURECOM
2229 route des crêtes, B.P. 193,
06904 SOPHIA ANTIPOLIS CEDEX, France.
Tel: 93.00.26.43, Fax: 93.00.26.27, email: sidou@eurecom.fr

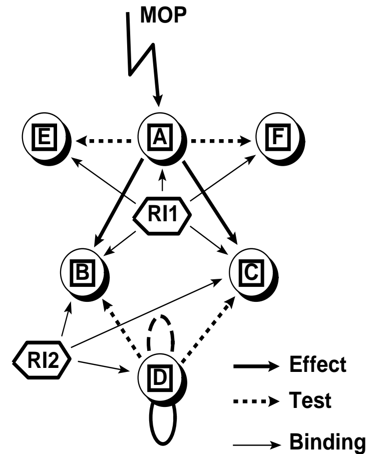
September 17, 1995

Abstract

In the context of the TIMS[1] project, a rapid-prototyping environment for TMN-systems is under development. The information modeling falicility is based on a fully object oriented framework (GDMO / ASN.1 support) extended with relationships (GRM support). Managed Object behaviors are specified thanks to the Behavior Language (BL), which enables both imperative and declarative specifications, in the context of relationships. Hints are given to show how policy objects and the so-called management policy scripts could be realized thanks to the provided information modeling facilities along with the BL. Finally, various semantics for the BL are described. Some of them constitute true behavioral simulation and analysis tools, which should reveal very useful in the context of detecting management policy conflicts.

[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, project F&E-288.

Relationship-based Behavior Formalization



Relationship-based Behavior Formalization Behaviors represent general dependencies between MOs, e.g. how one MO influences other MOs, or general properties valid among well defined MO sets. It is natural to think about such dependencies in the context of relationships. Relationships provide the means to specify cleanly the different neighboring contexts in the scope of which behaviors should manifest themselves. Relationship classes give, in their turn, the means to define the different neighboring flavors of interest for each MO instance. Then, within this framework, it is very straightforward to formalize how a MO change should affect its neighbors. This corresponds to triggered behaviors, where a triggering context determines in an imperative fashion when a behavior should be executed. On the other hand, being able to specify general dependencies among the MOs participating in a given relationship without any reference to any triggering context is also very interesting. The resulting so-called untriggered behaviors are thus specified in a purely declarative fashion.

Behavior Simulation Principle Behaviors manifest themselves as the propagation of effects among managed objects. Such behavior effects can be realized as (i) a MIB alteration (a modification of an attribute value or a MO creation or deletion), (ii) an emission of a notification, and (iii) an action being executed.

The execution of all these kind of effects may be subject to the fulfillment of some testing conditions on attribute values of associated MOs.

1. **Triggered behavior** : a management operation or a real resource change is applied on object **A** whose behavior exercise itself as side effects onto objects **B** and **C**, if additional testing conditions related to object **E** and **F** are fulfilled.
2. **Untriggered behavior** : object **D** depends on objects **B** and **C** : local side effects are propagated on **D**, if additional testing conditions on **B** and **C** are satisfied.
3. **Relationship context** : In any cases the behaviors are specified in the context of a relationship. That means that for the triggered behavior a relationship instance associating {**A, B, C, E, F**} has to exist. In the same way, for the untriggered behavior, a relationship instance associating {**B, C, D**} has to exist.

Formalization

```
(define-behavior
  (scope rel "ServiceResource-SubNetworkConnection-Nctps")
  (trigger (role "srv") (interface mgmt)
           (op update "lifeCycleState"))
  (pre (every (lambda (nctp-elem)
                (equal? (mo-send get (moi nctp-elem)
                                     (role "nctp")
                                     (attribute "lifeCycleState"))
                        "inService"))
              (ri-send binding-values (role "nctp"))))
  (body (mo-send action "activateSubNetworkConnection"
                 (interface mgmt) (mode confirmed)
                 (moi (ri-send binding-value (role "srv")))
                 (argument (ri-send binding-value (role "snc")))))
  (post (equal? (mo-send get
                         (moi (ri-send binding-value (role "srv")))
                         (role "srv") (attribute "lifeCycleState"))
                "inService")))
```

The general form of a simulated behavior is composed of :

1. **scoping context** : the relationship whose instances are concerned by the behavior.
2. **triggering context** : present only for triggered behaviors, specifies precisely the event (management operation or real resource change) that triggers the execution of the behavior (attribute change, object creation, object deletion, action). Since the scoping context is a relationship, it is also necessary to specify to which participant role this event is applied.
3. **pre-condition** : testing condition upon the MIB state, to be satisfied in order to launch the behavior body.
4. **body** : the body of a behavior is an imperative / procedural piece of *Scheme* [Ce91] code. There is no a priori structure imposed on it. Tests and effects are intended to be used in this body, reflecting the dependencies between MOs. For instance an object may be updated, created or deleted according to the presence, absence or the state of another object. Since usual programming features (i.e. control flow structures, variable notation...) are required at this level, the use of an existing and well-known programming language is a reasonable choice. This is also the approach taken in the DOMAINS project [FHHS93], which uses Eiffel construct to formalize behavior bodies. Though not mandatory, chosing the programming language of the targetted runtime environment may reveal very interesting in order to provide executable specifications and by this means facilitating the integration of behaviors in the runtime environment.
5. **post-condition** : testing condition upon the MIB state, to be satisfied at the completion of the execution of the behavior body.

Summary : the BL framework is defined by the unique *define-behavior* construct, inside of which usual *scheming* can be exercised. The *scheming* environment is extended with the APIs giving access to the entities of the currently supported information models, i.e. GDMO, ASN.1 and the GRM [Grm]. Note that, the kernel of the system could be ported to other information models by just providing the required APIs, as soon as all remains in the context of objects and relationships. The key design of the BL was to embed it in a programming environment, to ensure executability. Thus, the notation is deliberatly based on the *Scheme* language. The motivations for this choice are multiple. First the language itself is small and simple (syntax), very powerful, extensible through high and low level macros and interpreted which is ideal for rapid prototyping. To get more insight about the environmental issues, the reader can refer to [SME95].

Management Policies Representation

- Passive parts (domains) : relationships $\simeq$
  general MO grouping facility.
  GRM enables dynamic binding, nothing prevents from meta-relationships (hierarchies).
- Active parts (goals, constraints and modalities) : behaviors.
  authorization, permission (triggered behaviors),
  obligations, motivations (untriggered behaviors).
- Policy conflicts detection : **non determinism support**.

Domains and Relationships Domains are powerful means to specify the set of objects or member objects (subjects and target objects) to which a policy applies. Objects should always be created within a domain, which can also be represented as an object, in this way permitting domain hierarchies and subdomains. However, note well that it is not the objects themselves that are "physically" included in a domain object; it is rather the domain object that maintains references to its member objects. This obviously incurs domain overlaps of subjects, target objects or both. Consequently policy overlaps may occur which may result in conflicts. Thus, domains are no more than specific relationship instances whose description could be also given with pure relationship oriented models such as the GRM. The GRM provides notational tools for specifying relationships, their actual representation in the MIB and their management. A management relationship is defined as a collection of managed objects together with an invariant referring to the properties of the managed objects (participants). Management relationships are additional information modeling concepts but are represented and manipulated by existing facilities of the management information model [Mim]. At the abstract level, the realization independant properties are modeled (roles, behaviors, operations, notifications, inheritance, and qualifying properties). Then at the relationship mapping level, how the relationship features are mapped onto actual MOs and existing management operations is described. Therefore, relying on a generic relationship facility to model management domains seems to be a reasonable option.

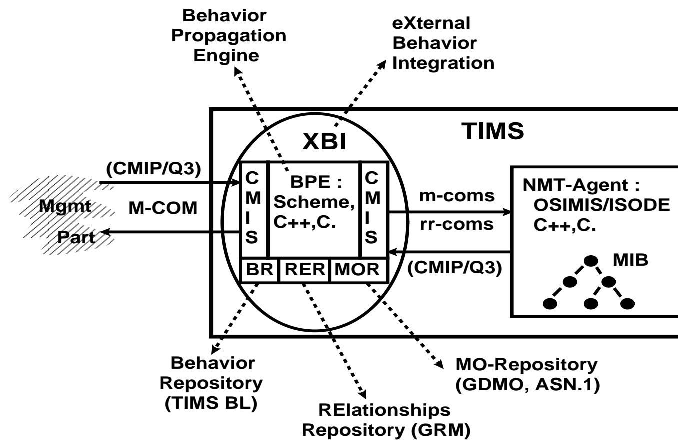
Policy modality, goals, constraints and Behaviors Then, for the more active features of policies, such as goals and constraints [SVMT93, Moffet94] our proposal is to consider their low level representations (management scripts) as management behaviors attached to the relevant subjects and target objects.

Permissions / authorizations can be integrated as triggered behaviors on the target objects, since they consist of properties to be checked (access rights...) upon the invocation of a management operation. Triggered behaviors can also be used to specify event handlers, like it is done in the DOMAINS Management Language [FHHS93].

In contrast obligations / motivations can be naturally represented with untriggered behaviors, since they correspond to behaviors to be launched by themselves as soon as the need arises.

Conflicts and Non Determinism The problem of policy conflicts detection is reduced to the problem of behavior analysis of non determinism. This is justified below as well as the proposed techniques of behavior analysis.

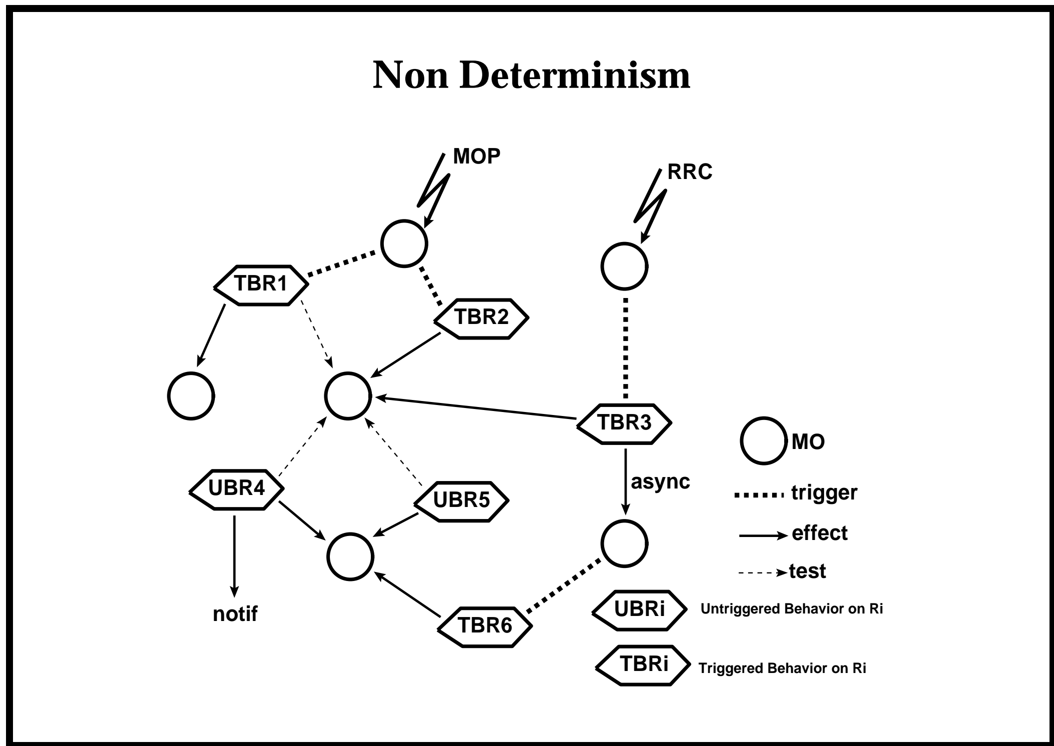
Rapid Prototyping and Simulation Environment



eXternal Behavior Integration (XBI) The eXternal (to agent toolkit) Behavior Integration (XBI), uses an agent toolkit for all the nice features provided, and are thus not to be to re-developed (e.g. CMIS processing, MO data store, SMFs. . .). The behaviors themselves are integrated externally, in an environment of our choice, which interoperates with some kind of dummy agent through the standardized CMIS interface. *Scheme* has been chosen for the current XBI implementation because of its simplicity, consistency and because behaviors can thus be interpreted, which ensures maximal flexibility, for rapid prototyping.

Behavior Propagation Engine (BPE) is the heart of the system since it defines the operational semantic of the Behavior Language (BL), which is not an obvious task since the BL is an hybrid formalism enabling to combine both imperative and declarative constructs. However, the more elaborated the semantic of the Behavior Language, the more powerful the sytem should be in terms of behavior analysis capabilities. As a consequence, a powerful semantic is a key feature in order to effectively test and exercise the consistency and robustness of Management Policies on prototype implementations before their introduction into the real world.

Salient Features of the BPE The BPE is initially solicited through the XBI interface to either invoke a management operation or to signal a change related to an underlying resource. Then the BPE basically works as a forward chaining inference engine, propagating the initial solicitation according to the behaviors in effect, until saturation, that is until the system reaches a new steady state. At any given time, a snapshot of the system is defined by the set of managed objects and corresponding attribute values (the so-called Management Information Tree). Note that such a system fits naturally into the Attributed Graph Rewritting System framework [Schuerr90, ZS91], where graph rewritting rules are no more than the specified behaviors. In such systems, non-determinism manifests itself if from the same inputs the processing can lead to different graph states according to the choices made at some intermediate step.



Non determinism introduced by the lack of synchronization In distributed systems, any variation in the timing of concurrent processings may causes them to access shared resources in different orders. Moreover, asynchronism is inevitable in such systems, since the occurrence of an event can come from the environment at any step of processing. Even, management operations themselves can be invoked asynchronously (for instance CMIS allows unconfirmed Set and Action services). Although, in the context of distributed and concurrent systems non determinism is an intrinsic property, the lack of synchronization is an error that can result in additional unwanted non determinism. Thus, it may be desirable to identify any sources of non determinism and to provide the relevant associated information in a suitable manner.

Non determinism introduced by the ambiguities in the system specification itself Declarative specifications (untriggered behaviors) is obviously a factor of non determinism. They may result as several behaviors to be propagated at a given stage of a behavior propagation, or as several management policies to be exercised at a given stage by one or several management authorities. If only one management authority is concerned the problem is simpler but may still result as conflictual situations depending on the chosen order of execution of the policies. If several management authorities are concerned, the problem is again more complicated since conflicts introduced by concurrency are then also cumulated. Since a behavior is naturally specified in the context of a relationship, it is possible to have several behaviors to be launched just because a MO is participating into several relationship instances of several distinct relationship classes. This may correspond, for example, to several access control policies defined for different contexts (represented as participations into relationships). This conflictual situations can be qualified as semantical conflicts.

Non determinism and Policy conflicts Management policy analysis and in particular conflict detection is considered as the support for non-determinism within the system. Management policy conflicts resulting from the concurrent processing of distributed management authorities may lead to inconsistencies and even catastrophic situations. In the same way, erroneous or ambiguous attribution of management roles to the different management authorities within an organization may result as semantical conflicts such as conflicts of interests..., that may reveal very harmful for its overall efficiency.

Behavior Analysis

- **Dynamic** versus Static (complementary techniques).
- Limited to the detection of errors.
- Can not be used to guarantee safety / liveliness properties.
- Closer to the real world.
- Should gain to be coupled with test generation.

Behavioral Analysis Techniques of non deterministic systems are usually classified as dynamic, if they involve process execution, and as static otherwise. Dynamic analysis aims at showing the presence of faults based on the implementation, or at least, on an executable specification; while static analysis aims at verifying the absence of faults based on a more likely non executable abstract model of the implementation. Since the BL is intended to provide executable specifications, it is natural to try to use the associated simulation environment as a tool for dynamic behavior analysis. Note well that, dynamic analysis can be used to show the presence of errors in the tested execution paths, but can never guarantee their absence in the general case. Although one can think about exhaustive dynamic, which could provide absolute certainty about the correcteness of a systems behavior, it is generally not feasible and too expensive. Though static behavior analysis techniques are intended to prove stronger properties, they are limited by their underlying representation model (e.g. temporal logic, Petri-Nets, Process Algebras. . .), which abstracts from most of the important details which can be expected to remain present within a running system. That is the reason why, dynamic behavior analysis techniques are complementary tools in the sense that problems raised should reveal closer to the real world than what could be achieved with their static counterparts. Finally, in order to get a sufficient level of confidence about the system, one can naturally think about coupling dynamic analysis with a test generation tool. This test generation would enable to submit more test cases and may be useful to select the more interesting test suites to be explored for behavior analysis.

Dynamic Behavior Analysis

- Principle : adopt an interleaving model to explore / linearize the behavior propagation space.
- Atomic steps : inter MIB interactions, need for behavior instrumentation.
- Conflict Detection Criteria : saturation states, BP profiles. . . .
- Requires both control and data backtracking.

Dynamic Behavior Analysis Principle Since the behavior simulation environment is able to track all the non deterministic choices made during the behavior propagation process, the idea is to adopt an interleaving model to explore the behavior propagation space. This is analog as what is done in static analysis based on process algebras – [Milner89] CCS, [Hoare85] CSP and [BK85] ACP – for parallel composition : concurrent events are reduced to traces of sequential events assuming that concurrent events can be linearized in any arbitrary order.

Events are determined clearly by MIB interactions that delimit the atomic portions of behavior execution flows that are subject to linearization in any relevant order. Within the BL, such execution flows can be easily isolated, since they correspond to the behavior processing in between two consecutive invocations on a MO (encapsulated into the `mo-send` BL construct). Since BL specifications are "parsed" in the sense that `mo-send` primitives are treated as *Scheme* language syntax extensions (macros). The resulting native *Scheme* behavior code actually executed, can be instrumented as needed. This way, flows whose relative execution order is not determined can be interleaved in any relevant order, defining the possible linearizations.

Conflict Detection Criteria A reasonable "objective" criteria that can be used in order to signal potential sources of conflicts is the multiple and different result criteria. In effect, if in case of non-determinism, at the potentially numerous behavior propagation saturations explored, the resulting states of the MIB are as well multiple, then the behavior specifier can be advised and after further analysis of the different behavior propagation paths, the original causes of non determinism can be isolated, the corresponding errors (resp. ambiguities) corrected (resp. raised). Resulting MIB states are considered to be equivalent, if they exhibit the same set of variables (MO instances along with attributes) with the same values. However, this is not a suficient condition, since during a behavior propagation path notifications may also be emitted. Thus, equivalence of behavior propagations should include not only the check for the same set of variables with the same values, but also the check of the same set of notifications emitted. If both sets match, then the corresponding behavior propagations may not be the subject of further analysis for the purpose of conflict detection. This criteria is objective in the sense that if two saturation states differ, further analysis should be exercised. Though, this does not identify all kind of races that may exist in the exercised propagations, this raises at least the more critical, giving up the more benign races [HM94]. In particular, one may be interested to treat the notifications as an ordered set (i.e. a sequence) rather than an unordered set, because this may reveal very important to deal with e.g. event correlation.

However, even if the BPs are equivalent in terms of same MIB reached at saturation state, one can also be interested in more refined criteria such as the relative length of BPs. This may reveal for instance useless computations and their corresponding interesting optimizations or on the other side erroneous short cuts.

Control and Data Bactracking Going through all the expanded linearizations requires both control and data backtracking capability. The more simple forms of control backtracking are based on recursive procedure calls supported by the execution stack. More elaborated forms of control backtracking can also be envisioned such as jumps through any execution stack contexts. Though small and simple, the used *Scheme* [Ce91, SF90] programming language provides in the language itself a powerful concept (continuations) and an associated construct (`call-with-current-continuation`) enabling to set up and memorize an execution context and the way to go back to this context, by simply calling it as if it was a usual function. Continuations are extremely useful for implementing a wide variety of advanced control structures, such as complex forms of backtracking.

Data backtracking is also an important issue. One can think about it in terms of transaction processing in data base systems. 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.

On the other hand, and in order to avoid the usage of a time consuming database management system, it is possible to rely on an undo + backtracking feature (also called data backtracking) mechanism which should be more efficient. Such a mechanism has been implemented in the PROGRES [Schuerr90, ZS91] environment, where a data backtracking facility enables to restore any previous arbitrary graph state. This facility is provided thanks to the GRAS system (GRAph Storage) [KSW92], which provides some kind of non-standard database management on top of which PROGRES acts as a manipulation and query language. GRAS offers sophisticated undo / redo mechanisms based on forward / backward executable graph deltas. Every graph modifying operation is stored together with its reverse operation in a command list. To retrieve a previous graph state, this command list is traversed backwards executing the reverse operations. Thus, the cost of data backtracking is proportional to the modifications made and not to runtime.

Semantic for Single Order BP (SO-BPE)

Algorithm 1 bpe:tbp(msg) :

1 ribks ← fetch-behaviors(msg)
2 bpe:execute(msg)
3 ∀ <ri, block> ∈ ribks :
4   **if** triggered?(beh(block))**then**
5   eval(body(block), ri, msg)
6   check-post(post(block), ri, msg)
7   **else**
8   eval(body(block), ri)
9   check-post(post(block), ri)
10   **end if**

Single Order Semantic This algorithm, for the BPE, defines a low level semantic for the BL, in the sense that propagation order choices made in cases of non determinism are hard-coded inside the BPE algorithm itself. Though this semantic presents not too much interest for behavior analysis, it is still useful to show the overall structure of the BPE. Note well however, that if we assume that the "∀" construct of instruction 3 goes through the matched behaviors to be propagated in a random order, then by playing the behavior simulation several times one can get some valuable insights about potential non determinism in the behavior specification. A Randomized Single Order operational semantic is thus defined by the algorithm that may be called the RSO-BPE algorithm.

"fetch-behaviors" can be detailed as follows :

Algorithm 2 fetch-behaviors(msg) :

1 { <ri, block>, ri ∈ parties(moi(msg)) ∧   ▷ *party(moi) = <role, ri>*
2   ∃ beh ∈ br, block ∈ blocks(beh) :   ▷ *br : behavior repository*
3   rel(ri) = rel(ctx(beh)) ∧ {   ▷ *blocks(beh) : pre-body-post behavior blocks*
4   { untriggered?(beh) ∧ firable?(block, msg) ∧ check-pre(pre(block), ri) } ∨
5   { triggered?(beh) ∧ check-pre(pre(block), ri, msg) }}

In any case the full single order semantic is still useful because it is the one that is used in the early stages of the rapid-prototyping process, where there is not yet any need for a full fledged behavior analysis semantic, and which would even be painful at this stage of development.

One should notice that for the invocation of triggered behavior bodies, the triggering message sent is also provided for the actual execution. This enables, for instance, to test the CMIS parameters given to the original management operation. For a `M_Set`, the new value can be tested in the pre-condition to specify behaviors associated to state transitions on a variable.

For untriggered behaviors, since there is no explicit trigger that enables to determine when to execute them, this is supported internally by the system thanks to the function `firable?`. The principle is to see if the operation invoked on a given object in a given role requires execution of the untriggered behavior, i.e. in order to maintain things consistent. For instance, if a variable is modified, and if the behavior makes in its body a `get` on it, the behavior is considered as firable. Obviously, this is an approximation, in many cases this re-execution is not necessary. However, a mechanism has to be provided to effectively propagate untriggered behaviors in the system.

Note that, the triggered behavior propagation algorithm mentioned above is implicitely recursively called in any circumstance where a new message is sent to a MO during the evaluation of a behavior body (see instruction 5).

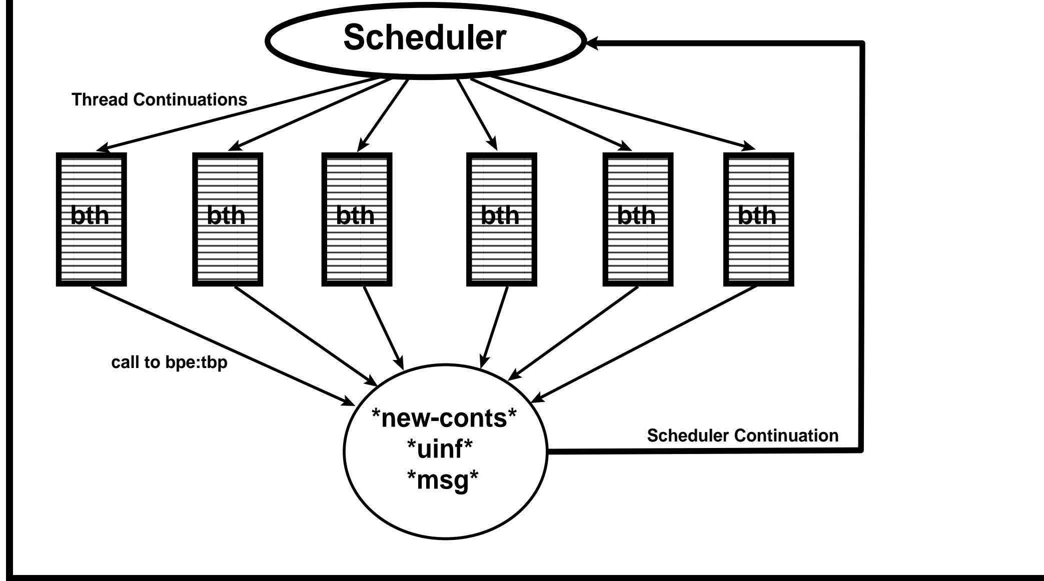
Semantic for Order Sensitive BP (OS-BPE)

Algorithm 3 bpe:tbp(msg, bag, trace) :

1 ribks $\leftarrow$ fetch-behaviors(msg)
2 uinf $\leftarrow$ bpe:execute(msg)
3 **if** empty?(ribks) $\wedge$ empty?(bag) **then**
4   dump-saturation(trace . msg)
5 **else**
6     $\Omega \leftarrow$ permutations(ribks)
7     $\forall \omega \in \Omega$:
8   bag $\leftarrow$ append(ω, bag)
9   eval-block(car(bag),
                (cdr(bag),
                trace . msg))
10 **end if**
11 undo(uinf)

Semantic for Order Sensitive BP produces linearization of behavior execution orders. This can be used to emulate the execution of management policies depending on a single management authority. Eventual non determinism and possible resulting conflicts are in this context relative to the different execution orders of the whole behaviors themselves. Thus, the unit for execution atomicity corresponds to entire behavior evaluations. Within this semantic, there is no provision for a more fine grain linearization that would enable to emulate for instance multiple management and tackle conflicts due to concurrent accesses to shared resources (MIB MOs). However, this semantic can be used to raise conflict originating in specification ambiguities.

Semantic for Concurrent Sequential Behaviors (Principle)



Semantic for Concurrent Sequential Behaviors enables to deal with full non determinism support for both specification ambiguities and Concurrency. This is possible thanks to the use of *Scheme* continuations. `call-with-current-continuation` is extremely useful for implementing a wide variety of advanced control structures, such as complex forms of backtracking.

The powerfulness of *Scheme* continuations enables to implement the linearization at the lowest level, i.e. at the MIB interaction level. Thus the atomicity for behavior execution flows is determined by the flow executed in between two consecutive MIB interactions of a given behavior thread.

However, this technique has to be used carefully since the combinatory resulting from the linearization can reveal prohibitive, depending on the amount of non determinism itself and the number of behavior propagation steps that are to be encountered in each behavior propagated. Both of these factors can not be known in advance, since purely execution dependent.

Complexity Analysis If we consider the number of linearizations of n threads of m intermediate steps each other, the worst case is to consider that the n threads are independant. Thus, each one can be scheduled at any location in $n \times m$ steps of each linearization. In actual executions, additional constraints are normally introduced between the steps of the different threads. Without such constraints, the total number of linearizations obtained is :

$$\prod_{i=1}^{i=n} \mathrm{C}_{i \times m}^{m} = \frac{(n \times m)!}{(m!)^n} \tag{1}$$

In the general case of n threads of respectively $m_1, m_2 \cdots m_n$ steps, this leads to $\sum_{i=1}^{i=n} m_i$ steps per linearization. Finally, the total number of linearization obtained is :

$$\prod_{i=1}^{i=n} \mathrm{C}_{\sum_{j=i}^{j=n} m_j}^{m} = \frac{\left(\sum_{i=1}^{i=n} m_i\right)!}{\prod_{i=1}^{1=n} m_i!} \tag{2}$$

Justification : All the linearizations can be obtained by first selecting a thread of, say thread$_1$ of m_1 steps, there are $\mathrm{C}_{\sum_{i=1}^{i=n} m_i}^{m}$ ways to allocate its schedule. Then, for each previously found schedule for thread$_1$, it remains $n-1$ threads to be allocated in $\sum_{i=2}^{i=n} m_i$ steps. This process repeated iteratively leads to (2).

Obviously, in actual executions each thread do not manifest a single number of steps, and as a consequence the size of each linearization is also variable.

Here are some illustrating examples :

m_1	m_2	m_3	res
2	2	0	6
3	3	0	10
4	4	0	70
5	5	0	252
2	2	2	90
3	3	3	1680
4	4	4	34650

Semantic for Concurrent Sequential Behaviors (CSB-BPE)

Algorithm 4 bpe:tbp(msg) :

```
1  call/cc(λ(this-cont) {
2      ribks ← fetch-behaviors(msg)
3      *uinf* ← bpe:execute(msg)
4      *msg* ← msg
5      ∀ <ri, block> ∈ ribks :
6          *new-conts* ← *new-conts* ∪
7              make-thread(ri, block, msg)
8      *new-conts* ← {this-conts} ∪ *new-conts*
9      *sched-cont*() }
```

Algorithm 5 bpe:schedule(conts, trace) :

```
1  if empty?(conts) then dump-saturation(trace)
2  else ∀ c ∈ conts
3      call/cc(λ(x) {
4          *sched-cont* ← x
5          c() }
6      let uinf ← *uinf*
7      schedule(conts \ {c} ∪ *new-conts*,
           trace . *msg*)
8      undo(uinf)
9  end if
```

The scheduler is passed as arguments a list of continuations (`conts`) to linearize. This represents the current execution context of the set of behavior threads to be run concurrently. The `trace` argument represents the trace of the BP leading to this set of continuations. The principle of the scheduler is quite simple, it launches iteratively the next execution step of each continuation (see algo. 5 instr. 5), just before this launching, it sets up a continuation for itself (see algo. 5 instr. 4) in a global variable `*sched-cont*` so that the scheduled thread is able to return to the scheduler. This way, the examination of the remaining threads can be processed (see algo. 4 instr. 9). Just before returning, the scheduled behavior thread sets up in global variables :

- the eventual new continuations that could have been created (`*new-conts*`).
- the undo informations (in *uinf*) that are to be undo-ed (see algo. 5 instr. 8) before the examination of the other threads.
- and finally the message executed so that to the new schedule invoked (see algo. 5 instr. 7), is passed a trace enriched with the last MIB interaction done. This way, at saturation, the

behavior propagation of the linearization reached can be dumped (see algo. 4 instr. 1) for further examination, if needed.

Implementation Issues

- Control backtracking.
- Data backtracking.

Related Work

- Advanced Debugging.
- Problem solving.

Control backtracking Simple forms of control backtracking use the execution stack and recursive calls (see algo. 3). However, more complex forms of backtracking often need features such as `continuations` which enables to set up any execution context so that bactracking onto it can be achieved (see algo. 4).

Data Backtracking This requires an undo facility, enabling all the MIB alterations to be undo-ed before backtracking. This is possible if enough info is kept about the state before any MIB alteration. Moreover, there is no problem associated to protections (read-only variables. . .) since the simulation environment already provides some kind of physical access to any MOs through a so-called real-resource interface.

Related Work (Problem Solving) PROGRES (PROgrammed Graph Rewriting Systems) [Schuerr90, ZS91] is a high level multi-pardigm language for the specification of complex structured data types and their operations. The basic programming constructs are graph rewriting rules (productions and tests) and derived relationships on nodes (paths and restrictions). Then basic operations may be combined to build partly imperative, partly rule based complex graph transformations. Due to the problem of non determinism already mentioned, the semantic of PROGRES involves both control + data backtracking. Data backtracking is achieved through an undo facility on attributed graphs. PROGRES works also as a forward chaining system, however, the key difference of such a problem solving environment with respect to a behavior simulation environment is that the inference engine is given a goal to achieve, which is explicitely stated. In this context, backtracking occurs when dead-ends or some constraint is violated during the search process. A dead-end is equivalent to a saturation state that is not satisfactory with respect to the targetted goal. Constraint violation is used to drive the search process, that is for pruning un-relevant search branches based on assertions about the user problem's domain. For the purpose of behavior simulation, if different saturation states are reached, what is interesting is to analyse them to check if they correspond to the management policies in application. On the other hand, if an assertion (pre / post condition) is violated, this raises explicitely an invalid management policy or at least an incorrect translation of a policy into the low level behavior scripts. Though not directly portable in the context of behavior simulation, the significant work accomplished

by the PROGRES team at University of Aachen, states clearly the underlying techniques of interest, namely control / data backtracking in the context of attributed graphs rewriting systems.

Related Work (Program Debugging) An execution backtracking facility in interactive source level debuggers allows users to mirror their though processes while debugging [ADS90]. This enables to work backwards from the location where an error is manifested and determine the conditions under which the error occured. 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). In effect, current so-called "high level" source debuggers 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 [ADS90], data backtracking is also handled thanks to undo mechanisms working on usual programming data structures (records, arrays. . .). The specific problem of input / output operations is well stated. In effect I/O operations are in general not undo-able (e.g. characters sent to a line printer, read / write on files. . .). One possible approach to solve this problem is to buffer I/Os and be able to push back buffered I/Os. Another solution, applicable to file I/Os is to record file pointer offsets and `lseek` to restore previous file contents for backtracking. Note that in the management context, notifications present the same characteristics as I/Os, of not being undo-able, e.g. once sent to a management party. However, as soon as the system stays into a pure simulation environment for the pure behavior analysis phase, this problem is avoided. Coupling the system with e.g. a management part is obviously possible, but would be restricted to work with a single order behavior propagation such as the one presented in 2. However, if really needed, one can still think about a buffering strategy (based on relevant setup on event forwarding discriminators and discrimator constructs defined in [Ermf]), in order to make emittted notifications undo-able.

Conclusion

- Current implementation : Single-Order BPE.
- Further Issues :
  - Applicability and usefulness.
  - Coupling with test generation.
  - Use of static behavior analysis tools ?

Current Implementation Status The current implementation, made in the context of the TIMS project is limited to the single order behavior propagation engine described in 2. This implementation has been tested with a simple case study based on a real-life TMN environment (see [SME95] for more details). The simple case study is concerned with the application of the TIMS tool-set in the Optical Access Network Management (Fiber-in-the-loop concept), currently in the standardization process in ETSI TM2 [De tm2209]. This choice proved to be a wise one; (i) the team learnt more about the TMN-specific problems of Information Modeling and (ii) immediate feedback of our results into ETSI was possible. Thus, underlying concepts and environmental issues have been validated to a reasonable

extent.

Conclusion and Further Issues Only some hints are given towards the feasibilty of the integration of management policy concepts into the behavior formalization and simulation environment, in terms of both information modeling issues and more dynamic issues such as policy modalities, goals and constraints. Though, it is assumed, that the relationship oriented behavior formalization facilities described are powerful enough for this purpose; it would be useful to envision a case study to actually test the usability of the TIMS tool-set in the context of exercising and testing management policies in the early stages of their specification.

Possible extensions of the BPE were presented in this paper, they define much more powerful operational semantics for the advocated behavior language (BL), enabling to deal with non determinism which is argued as being directly applicable for management policy conflict detection.

Though the design of such semantics was clearly stated, as well as their nice features in terms of dynamic behavior analysis, their applicability and usefulness for the test of Management Policies remains to be proven, and in particular for the detection of conflicts between Management Action Policies.

Environmental issues are also very important, in particular the presentation / visualization of the different behavior propagations reached because of non determinism may reveal quite difficult in itself. But, such a facility is very useful for the purpose of analysis and explanation.

Coupling with test generation As stated before, a major drawback of dynamic behavior analysis techniques such as the ones presented, is that they are limited to the analysis of the submitted test cases. They can raise erroneous situations for these test cases but, can not prove more general properties about the whole specification such as the absence of deadlocks or the guarantee of some liveliness property. In this context, a test generation facility could be used to submit more test cases and in particular the more stressing ones, based on for instance, overlap analysis between policies seems to be a promising item for further studies.

Static Techniques Static behavior analysis techniques can be used to prove stronger properties from the specification itself, one can be tempted to map the BL to an abstract representation model used for static behavior analysis. As an intuition, process algebra notations (CCS, CSP, ACP) may be the more easy to use for this mapping, they provide a language based notation with specific operators in order to specify parallelism in the specification.

Acknowledgments The author would like to express his grateful thanks to Olivier Festor and to Sandro Mazziotta for their careful reading and valuable comments.

References

[ADS90] Agrawal (Hiralal), DeMillo (Richard A.) et Spafford (Eugene). – *An Execution Backtracking Approach to Program Debugging.* – Rapport technique n°SERC-TR-22-P, Software Engineering Research Center Purdue University, September 1990.

[BK85] Bergstra (Jan A.) et Klop (J. W.). – Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, vol. 37, n°1, mai 1985, pp. 77–121.

[Ce91] Clinger (W.) et (editors) (J. Rees). – $Revised^4$ Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, vol. 4, n°3, 1991. – Available through ftp on every scheme repository.

[De tm2209] Transmission and Multiplexing : Operations and Maintenance of Optical Access Networks, ETSI TM2, DE-TM2209, STC Draft 08d, 16.9.1994.

[Ermf] Systems Management - Part 5: Event Report Management Function, ISO/IEC 10164-5, ITU X.734.

[FHHS93] Fischer (Axel), Herpers (Martine), Holden (David) et Sievert (Stephan). – The DOMAINS Management Language. *In : Integrated Network Management III (C-12)*, éd. par Hegering (H.-G.) et Yemini (Y.). ©IFIP, pp. 181–192. – Elsevier Science Publishers B.V. (North-Holland).

[Grm] 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.

[HM94] Helmbold (D. P.) et McDowel (C. E.). – *A Taxonimy of Race Conditions.* – Rapport technique n° UCSC-CRL-94-34, University California, Santa Cruz, sept 94.

[Hoare85] Hoare (C. A. R.). – *Communicating Sequential Processes.* – Prentice-Hall, 1985.

[KSW92] Kiesel (N.), Schuerr (A.) et Westfechtel (B.). – *Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications.* – Rapport technique n°92-44, Technical University of Aachen (RWTH Aachen), 1992.

[Milner89] Milner (Robin). – *Communication and Concurrency.* – Prentice-Hall, 1989.

[Mim] Management Information Services - Structure of Management Information - Part 1: Management Information Model, ISO/IEC DIS 10165-1, ITU X.720.

[Moffet94] Moffet (Jonathan D.). – *Network and Distributed Systems Management*, chap. 17, Specification of Management Policies and Discretionary Access Control, pp. 455–481. – University of York, jdm@minster.york.ac.uk, Addison-Wesley Publishing Compagny, 1994.

[Schuerr90] Schuerr (Andy). – *PROGRESS: A VHL-Language Based on Graph Grammars.* – Rapport technique n° 90-16, Technical University of Aachen (RWTH Aachen), 1990.

[SF90] Springer (G.S.) et Friedman (D.P.). – *Scheme and the Art of Programming.* – MIT Press, 1990.

[SME95] Sidou (Dominique), Mazziotta (Sandro) et Eberhardt (Rolf). – TIMS : a TMN-based Information Model Simulator, Principles and Application to a Simple Case Study. *In : Sixth International Workshop on Distributed Systems : Operations & Management.* IFIP / IEEE. – Ottawa - Canada, 1995.

[SVMT93] Sloman (Morris S.), Varley (Barry J.), Moffet (Jonathan D.) et Twidle (Kevin P.). – Domain Management and Accounting in an International Cellular Network. *In : Integrated Network Management III (C-12)*, éd. par Hegering (H.-G.) et Yemini (Y.). ©IFIP, pp. 193–206. – Elsevier Science Publishers B.V. (North-Holland).

[ZS91] Zuendorf (Albert) et Schuerr (Dr. Andreas). – *Nondeterministic Control Structures for Graph Rewriting Systems.* – Rapport technique n°91-17, Technical University of Aachen (RWTH Aachen), 1991.