The Semantics of COALA in CO-OPN/2*

Julie Vachon

LGL-DI, Swiss Federal Institute of Technology
CH-1015 Lausanne, Switzerland
fax:(41)(21) 693-5079
Julie.Vachon@epfl.ch

1 Introduction

This report presents the formal semantics of COALA, a design language based on the concept of Coordinated Atomic Actions (CA action). COALA has been developed with the intent of providing a concrete language to express CA actions and thus facilitate the design of softwares whose structure relies on this concept.

CA actions have been one of the main research topics of the former DeVa Esprit Project. A paper has been published [VBB+99] which summaries the principles of CA Actions and introduces the syntax of COALA as well as a short overview of its semantics. As a complement to this paper, this report concentrates on the formal semantics of the language and provides all the details. The semantics is expressed using the formal object-oriented specification language CO-OPN/2. The complete semantics is given in the appendix; specifications were checked (syntax and types) and formatted using the tools developed for CO-OPN/2[1].

The following subsections should help the reader understanding the context and the objectives of this work. The main principles of CA actions and COALA are first recalled, the needs for a formal semantics are then exposed and finally the use of CO-OPN/2 as a target language describing the semantics is justified.

1.1 COALA and CA-Actions

Coordinated Atomic Actions

The Coordinated Atomic Action (or CA action) concept ([XRR+95], [RRS+97a], [RRS+97b]) is a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting objects in distributed object-oriented systems. A CA action provides a mechanism for performing a group of operations on a collection of (local or external atomic) objects. These operations are performed cooperatively by one or more *roles* executing in parallel within the CA action. The interface to a CA action specifies the objects that are to be manipulated by the CA action and the roles that are to manipulate these objects. In order to perform a CA action, a group of

*This work was partially supported the ESPRIT Project LTR DeVa, no. 20072
[1]Tools are available at the following URL : http://lglwww.epfl.ch/Conform/soft.html

execution threads must come together and agree to perform each role in the CA action concurrently, with each thread undertaking its appropriate role.

CA actions present a general technique for achieving *fault tolerance* by integrating the concepts of conversations (that enclose cooperative activities), transactions (that ensure consistent access to shared objects), and exception handling (for error recovery) into a uniform structuring framework. More precisely, CA actions use conversations ([Ran75]) as a mechanism for controlling concurrency and communication between threads that have been designed to cooperate with each other. Concurrent accesses to shared objects that are external to the CA action are controlled by the associated transaction mechanism that guarantees the ACID (atomicity, consistency, isolation, and durability) properties [GR93]. In particular, objects that are external to the CA action, and can hence be shared with other actions concurrently, must be atomic and individually responsible for their own integrity. In a sense CA actions can be seen as a disciplined approach to using multi-threaded nested transactions [Gro96], and to providing them with well-structured exception handling.

The CA action framework is based on a disciplined and rigorously-defined exception handling [XRR98] which relies on several important principles: involving all action roles in handling any exception raised inside action, associating exception propagation with the nested action structure and resolving concurrently-raised exceptions. When a single (internal) exception is *raised* by a role within a CA action, other roles of the same action are *informed* and all activate handler corresponding to this exception. If exceptions are raised concurrently within a CA action, an *exception graph* is used to determine which exception handler has to be activated by each role to recover the action (the exception graph imposes a partial order on all action exceptions in such a way that a covering, or, resolving exception can be found for any exceptions which can be raised concurrently inside the action). The CA action is completed with a *normal* outcome if all exception handlers are completed successfully, otherwise each handler *signals* a common (interface) exception to the enclosing CA action. Among these interface exceptions, *abort* and *fail* are two special exceptions which are respectively used to signal action abort (implies that the action has been aborted and all of its effect have been undone) or action failure (indicates that an attempt has been made to abort the action but it was not possible to complete it successfully).

COALA

As to outline a precise semantics of the CA actions and to allow complex concurrent systems to be formally designed and specified using CA actions, a new formal specification language called COALA (COordinated Atomic actions LAnguage) has been developed. This language includes a minimal set of instructions capturing the essence of CA actions; it has been created as an extension of the CO-OPN/2 language. Its semantics is also being expressed in the CO-OPN/2 specification language. COALA can thus benefit from the formal techniques and tools developed for CO-OPN/2 and use them to verify and test applications that are written in COALA and use CA actions ([BBP96], [PBB98]).

The syntax of COALA and the explanations of its instructions have been explained thoroughly in [VBB+99]. The reader is strongly encouraged to first have a look at this paper, which is an introduction to COALA. As a reminder,

Appendix A presents COALA's BNF grammar. Figure 1 recalls the basic shape of a COALA program.

CAA `CaaName;`

Interface

  Use ... ;; *COOPN/2 specification modules used.*

  Roles ... ;; *Declaration of roles.*

  Exceptions ... ;; *Declaration of external exceptions.*

Body

  Handlers ... ;; *Declaration of handlers.*

  Resolution ... ;; *Resolution tree for raised exceptions.*

  Role `role1;` ;; *Program which role1 has to execute.*
    Begin
    ...
    End

    Handler `H;` ;; *Program of handler H for role1.*
      Begin ...**End**
    End `H;`

  End `role1;`

  Role `role2;` ;; *Program which role2 has to execute.*
    Begin
    ...
    End

    Handler `H;` ;; *Program of handler H for role2.*
      Begin ...**End**
    End `H;`

  End `role2;`

End `CaaName;`

Figure 1: General structure of a COALA program

1.2 Objectives of the semantics

As the work on CA Actions and COALA progressed, needs for a clear and formal definition of the elaborated concepts became more and more necessary. This semantic work had thus two objectives

- Provide a formal document which would help understand and establish the basis of COALA's CA actions (principles and properties).
- Obtain a translation of COALA programs in a formal language which could allow (or at least facilitate) test and proof of properties. When appropriate tools will be available, this formal semantics can also be used for simulation and prototyping of COALA programs.

The semantics explained in this report aims at translating COALA programs in terms of CO-OPN/2 objects, which have the advantage of themselves having a formal and precise semantics on which we precisely intent to rely. These CO-OPN/2 objects are being formally specified by a set of behavioural axioms. The semantics of a CO-OPN/2 object is defined by a transition system describing all the object's admissible behaviours and only them. Hence, the set of CO-OPN/2 objects giving the semantics of a COALA program thus itself corresponds to a global transition system whose transitions define all and only the admissible (i.e. observable and expected) behaviours of the COALA program.

1.3 Motivations for the choice of CO-OPN/2

The object-oriented specification language CO-OPN/2 has been chosen as the target language of COALA semantics. CO-OPN/2[2] is a specification language based on high level synchronised Petri nets introducing object-orientation mechanisms. CO-OPN/2 is particularly well-suited for the formal modelling of concurrent object systems. It has been chosen as our target language for describing COALA's semantics because of its multiples advantages. It indeed offers some good structuring means, interesting concurrency modelling capabilities, useful atomicity properties, object-oriented features, etc. Above all, its formal foundations (i.e. Petri nets) allow for further verification capabilities based on the development and the use of formal methods and testing strategies. Choosing CO-OPN/2 as our basic framework to express COALA's semantics allowed us to benefit from all these advantages and to integrate them usefully in further projects.

To summarise, CO-OPN/2 not only defines a formal notation describing complex systems but is also a framework providing helpful formal development methods and tools. This formal framework actually includes checking tools (for syntax and types), a denotational semantics, a refinement theory, means for contextual coordination, testing tools and a prototyping methodology. Details about these aspects of CO-OPN/2 can be found in [Bib97], [BB97], [Ser99], [Pér98].

The following subsection should help the readers (who are not familiar with CO-OPN/2) understand the main elements of the syntax and the basic behaviours of this formal specification language.

Basics of CO-OPN/2

CO-OPN/2 (Concurrent O-O Petri Nets) is a formalism developed for the specification and design of large O-O concurrent systems ([Bib97] and [BBG97]). Such system consists of a possibly large number of entities, which communicate by triggering parameterised events. The events to which an object can react are also called its methods. The behaviour of the system is expressed with algebraic Petri nets. A CO-OPN/2 specification consists of a collection of two different kinds of modelling entities: algebraic abstract data types (ADTs) and classes. Algebraic sorts are defined together with related functions, in ADT modules. Algebraic sorts are used to specify values such as the primitive sorts (integer, boolean, enumeration types, etc.) or purely functional sorts (stacks, etc.). Class' type sorts are defined together with their methods in a distinct class module. Such a module corresponds to the notion of encapsulated entity that holds an

[2] CO-OPN/2 was developed at the Software Engineering Laboratory of the Federal Institute of Technology in Lausanne

internal state and provides the outside with various services. Cooperation between objects is performed by means of a synchronisation mechanism, i.e. each event may request synchronisation with the methods of one or of a group of partners using a synchronisation expression. Three synchronisation operators are defined: "//" for simultaneity, ".." for sequence, and "+" for alternative. The syntax of the behavioural axiom that includes synchronisation is

BehAxiom	::=	*Event* [**With** SynchroExpression] **::**
		[*Condition* =>] *Precondition* -> *Postcondition*

Condition is a condition on algebraic values, expressed with a conjunction of equalities between algebraic terms. *Event* is an internal transition name or a method with term parameters. *SynchroExpression* is the (optional) expression described above, in which the keyword **With** plays the role of an abstraction operator. *Precondition* and *Postcondition* correspond respectively to what is consumed and to what is produced in the different places within the net (in arcs and out arcs in the net).

2 The Semantics of COALA

2.1 General architecture of the semantics

This work on COALA's semantics constitutes the first step towards the implementation of an automatic translating tool of COALA programs into their formal description in CO-OPN/2. More precisely, this report proposes a set of CO-OPN/2 generic classes specifying the general behaviour of (1) CA actions, (2) Roles and (3) Internal/External objects accessed by CA actions. Each CA action, each role and each object of a COALA program is being semantically described by a CO-OPN/2 object, which is an instance of one of the generic classes mentioned above or is an instance of one of their more specific subclasses. Indeed, particular instances and subclasses must be created to express the precise semantics of a given COALA program. An automatic translation tool of COALA programs into CO-OPN/2 specifications should thus be responsible for creating the right CO-OPN/2 subclasses and object instances (using the generic classes proposed in this work) and for appropriately initialising them. Figure 2 shows an example of the class hierarchies (generic classes and specific subclasses) and the related object instances that such a translator would create.

Figure 3 introduces the main CO-OPN/2 generic classes used to built the semantics of COALA programs and illustrates the CLIENTSHIP dependency relation existing between these class modules. The relation $M \to M'$, represented by the arrows on the figure, indicates that class module M has a behavioural axiom requiring a synchronisation with a method defined in class module M'.

As mentioned, COALA's semantics is obtained using the definition of three main generic classes in CO-OPN/2. These class modules are the following:

- **Class** `Caa`. This class describes all the coordination activities managed by a CA action. This includes starting all roles synchronously, solving distributely raised exceptions and managing how they must be handled, coordinating ending roles, signalling exceptions and outcomes, etc.

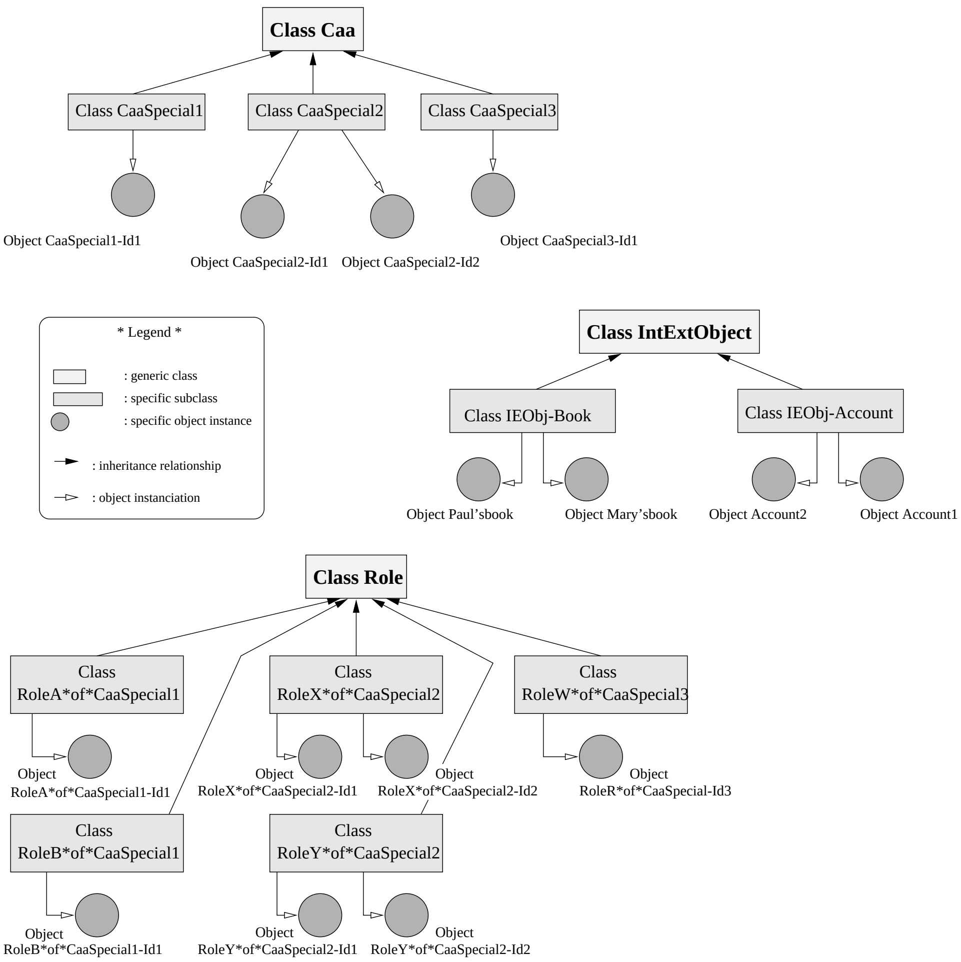

Figure 2: Inheritance hierarchies and object instances created by the semantic translator.

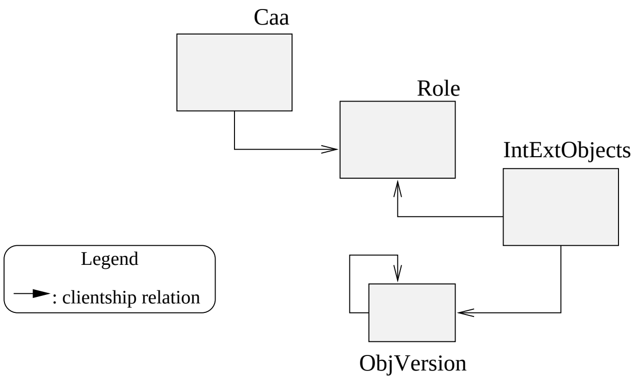

Figure 3: Clientship dependency relation between the generic classes of the semantics.

- **Class** `Role`. This module describes how a role is started, how it ends and how it interprets the instructions which it has to execute (sequential and conditional instructions, operations on ext./int. objects, exception raising, etc.)
- **Class** `IntExtObject`. This class describes how a COALA object processes the operation requests it receives, while together ensuring respect of the operations' ACID properties in the context of embeded CA actions. This class uses objects of another CO-OPN/2 class called `objVersion` to keep trace of the different object state versions that each CA action has at one time. More precisely, these object state versions represent the different views that each individual CA action (operating on the object) has of the object's state. When a CA action commits, the view it has of the object's state becomes the current state of the object. If it aborts, no change is made to the object's current state.

This object-oriented approach adopted to build COALA's semantics presents many advantages. Among others, this semantic definition appears to be modular, generic, clear and quite intuitive. These genericity and modularity qualities allow to translate COALA's CA actions, roles and internal/external objects into CO-OPN/2 objects by a simple instantiation mechanism, using the predefined CO-OPN/2 classes mentioned above. Indeed, classes `Caa`, `Role`, and `IntExtObject` have been defined to be as generic as possible. For that matter and as explained later on, class `Role` integrates a generic interpretor which allows the objects of this class to execute any sequence of instructions they have been instantiated with. This interpretation task is described in class `Role` by the axioms of spontaneous internal transition `eval`.
Remark: Indeed, other approaches could have been chosen. Among others, an approach by "compilation" was proposed. It consisted in constructing, at compilation time, specific CO-OPN/2 role objects, each having a different specification according to the particular sequence of instructions it had to execute. However, we prefered our object-oriented approach for its modularity and genericity properties, even though it required the specification of a generic interpretor.

2.2 Detailed description of CO-OPN/2 generic classes used in the semantics

2.2.1 Class `Caa`

A CA action is a special entity coordinating the execution of a certain number of threads each accomplishing a given task called a "role". A CA action has four main activities: (1)starting the roles, (2)collecting the roles' outcomes, (3)ending the execution of roles, (4)solving and handling the exceptions distributely raised by roles.

(1) *Starting the roles*
A CA action must synchronise all the calling threads that register for a role so that all roles are fulfilled and started at the same time. Internal transition `startRoles` accomplishes this task. For each thread calling a role (`c.callRole(r, extObj)`), a corresponding role object is started (`r.start`). It is only once all

the roles have been fulfilled that the token retrieved from place `StartR` is put in place `Run` hence allowing the CA action to go on with its other activities (cf. first axiom of `startRoles`).

```
startRoles :: =>
    StartR @, Roles [], CalledRoles rcalled rcalling ->
    Run @, Roles rcalled, CalledRoles rcalled rcalling;


startRoles With
    (c.callRole(r, extObj) .. r.start(extObj, caaObj)) :: =>
    StartR @, Roles (r ' rlist), Obj caaObj,
    CalledRoles rcalled rcalling ->
    StartR @, Roles rlist, Obj caaObj,
    CalledRoles (r ' rcalled) (c ' rcalling);
```

Remark: Since a *thread* usually calls a role while it is itself performing the role of another CA action, the object variable `c` representing the calling thread is thus also of type `role`.

(2) *Collecting the roles'outcomes*

The collection of the roles' outcomes is performed by transition `acceptTest`. While individual outcomes are gathered, the final outcome is being simultaneously calculated by the operation `result` (who's signature is `outcome, outcome -> outcome`). As soon as the collection of outcomes has begun, the token found in place `Run` is put in place `AcceptTest` (see first axiom) as a notification that the CA action has already started gathering up some of the roles' outcomes.

When all the roles' outcomes have been collected, the token is put into place `EndR` (see second axiom), thus allowing the termination of roles to begin...

```
acceptTest With r.outcome(outc) :: =>
    Run @, CalledRoles (r ' rlist) clist ->
    AcceptTest @, CalledRoles rlist clist, Outcome outc;

acceptTest :: =>
    AcceptTest @, CalledRoles [] clist, Roles rlist ->
    EndR @, CalledRoles rlist clist, Roles rlist;

acceptTest With r.outcome(outc1) :: =>
    AcceptTest @, CalledRoles (r ' rlist) clist,
    Outcome outc2 ->
    AcceptTest @, CalledRoles rlist clist,
    Outcome (result(outc1, outc2));
```

(3) *Ending the roles*

When roles are ended, the final outcome must be signalled to all the calling threads (`c.signalOutcome(outc)`). Moreover, according to the CA action's final outcome, each role is responsible for sending a *commit* or *abort* request to all the objects it has been operating on (cf. `r.commit` and `r.abort`).

```
endRoles :: =>
    EndR @, CalledRoles [] [] -> CAAFinished @;
```

```
endRoles With  r.abort .. c.signalOutcome(abort) :: =>
    EndR @, Outcome abort, CalledRoles (r ' rcalled) (c ' rcalling) ->
    EndR @, Outcome abort, CalledRoles rcalled rcalling ;

endRoles With r.commit .. c.signalOutcome(outc) ::
    not(eqOutc(outc, abort)) = true =>
    EndR @, Outcome outc, CalledRoles (r ' rcalled) (c ' rcalling) ->
    EndR @, Outcome outc, CalledRoles rcalled rcalling ;
```

(1) *Solving and handling distributely raised exceptions*

During their execution, roles can raise exceptions. When an exception is raised , the CA action takes it up (cf. first and second axioms of transition `getExcept`) and then looks at all the other roles to see if they also have raised an exception at that same time (cf. third axiom). An exception is being raised by a role `r` if `r.exRaised(e)` returns a non `nil` value in `e`. Remark that if a CA action has already started gathering up some roles' outcomes (a token is found in place `AcceptTest` instead of place `Run`) when an exception is raised, it must forget collected outcomes (the place `Outcome` is emptied, cf. first axiom) to start the exception solving process. A token, retrieved from place `Run` or `AcceptTest` as the case may be, is put in place `Suspended-Get`, thus preventing any other activity to start.

When all the exceptions raised at one time have been collected, the set of exceptions is solved. The solution (cf. `solve(i,exclist)`) in fourth axiom) is a unique global exception to be handle by each role. At this point, the token is moved from place `Suspended-Get` to place `Handle-Get` which therefore allows transition `handleExcept` to fire.

```
getExcept With r.exRaised(e) ::
    (not(eqExcept(e, nil))) and (r isIn? rlist) = true =>
    AcceptTest @, CalledRoles rlist clist,
    Roles orlist, Outcome outc ->
    Suspended-Get @, CalledRoles (remove r orlist)  clist,
    Roles orlist;

getExcept With r.exRaised(e) ::
    (not(eqExcept(e, nil))) and (r isIn? rlist) = true =>
    Run @, CalledRoles rlist clist ->
    Suspended-Get @, CalledRoles (remove r rlist)  clist;

getExcept With r.exRaised(e) :: =>
    Suspended-Get @, CalledRoles (r ' rlist) clist,
    ExceptRaised exclist ->
    Suspended-Get @, CalledRoles rlist clist,
    ExceptRaised (e ' exclist);

getExcept :: =>
    Suspended-Get @, CalledRoles [] clist,
    Roles rlist, ExceptRaised exclist ->
    Suspended-Handle @, CalledRoles rlist clist,
    Roles rlist, ExceptRaised ((solve(i,exclist))'[]);
```

As for transition `handleExcept`, it simply activates the handling mechanism of each role `r` for the global exception `e` found in the single element list of place

ExceptRaised. This is expressed by synchronisation `r.handle(e)` in the second axiom. When all roles have handled the exception, the CA Action returns to its normal state and the token is put back in place `Run`.

```
handleExcept :: =>
     Suspended-Handle @, CalledRoles [] clist,
     Roles rlist, ExceptRaised (e'[]) ->
     Run @, CalledRoles rlist clist,
     Roles rlist, ExceptRaised ([]);

 handleExcept With r.handle(e) :: =>
     Suspended-Handle @, CalledRoles (r ' rlist) clist,
     ExceptRaised (e'[]) ->
     Suspended-Handle @, CalledRoles rlist clist,
     ExceptRaised (e'[]);
```

2.2.2 Class `Role`

A role in a CA action is a particular program which a thread has to execute concurrently and cooperatively with other threads performing other roles in the same CA action. During the execution of a COALA role program, exceptions can be raised or signalled, operations on objects can be applied (ensuring ACID properties), roles in nested CA Actions can be called, etc. All these instructions can be executed in sequence or within the context of a conditional instruction.

Each object of class `Role` contains an instruction interpretor specified by an internal transition called `eval`. Instructions to be evaluated are put in place `Instr`. The evaluation process is controlled by a token put into place `Run` when it must be running, and transfered into place `Suspended` when it must temporarily be stopped (e.g. when a role raises an exception).

Before going on with the detailed explanation of the role program interpretor, let's mention that class `Role` also provides useful methods. These are invoked...

1. to get some informations about the role:
   - to get the role's outcome (`outcome _ : outcome`),
   - to get the exception it has raised (`exRaised _ : exception`),
   - to get the operation the role wants to apply to an internal/external object (`evtReq _ _ : event caaId`; `abort`; `commit`),
   - to get the role it has called in a nested CA action (`callRole _ _ roleType objlist`).
2. to give the role some informations :
   - to inform the role about the global exception it has to handle (`handle _ : exception`),
   - to signal the outcome of the role it has called in a nested CA action (`signalOutcome _ : outcome`).

Role Program Interpretor

The interpretation of an instruction is described by one or many axioms of transition `eval` in class `Role`. Here follows some detailed comments about the evaluation of each type of instructions that can be used in COALA role programs.

• Instruction <instr1> ; <instr2>
When a sequence instruction is found in place `Instr`, it is automatically split in two parts (cf. first axiom): the first part of the sequence stays in place `Instr` while the second part is put in place `InstrSnd`, waiting to be executed just afterwards (cf. second axiom). When both places contain the value `Iend`, this indicates that the execution of the role program is completely terminated (cf. third axiom).

```
eval :: =>
   Run @, Instr (Sequence i1 i2), InstrSnd i3 ->
   Run @, Instr i1, InstrSnd (Sequence i2 i3);

eval ::
   !(i = Iend) =>
   Run @, Instr Iend, InstrSnd i ->
   Run @, Instr i, InstrSnd Iend;

eval :: =>
   Run @, Instr Iend, InstrSnd Iend ->
   Run @, Outcome normal;
```

• Instruction `Call` <roleName>(e_1, ...e_n) `Of` <caaName>
To call the role of a nested CA action, a reference on the called role is put into place `RoleToCall` together with its arguments. An object of class `Caa` representing the nested CA action should eventually synchronise with the method `callRole` of the role to retrieve the arguments and start the called role of the nested CA action.

```
eval :: =>
   Run @, Instr (CallRole r withArgs args) ->
   RoleToCall r args;
```

• Instruction `If` <cond> `Then` <instr1> `Else` <instr2>
As usual, if expression `cond` evaluates to `true` in the given context `c`, the first branch of the conditional must be executed and instruction `i1` is thus put into place `Instr`. Otherwise, if `cond` evaluates to `false` in the same context, instruction `i2` is put in `Instr`.

```
eval ::
   (subst cond c) = true =>
   Run @, Instr (If cond Then i1 Else i2), Ctxt c ->
   Instr i1, Ctxt c, Run @;

eval ::
   (subst cond c) = false =>
   Run @, Instr (If cond Then i1 Else i2), Ctxt c ->
   Run @, Instr i2, Ctxt c;
```

• Instruction `Raise` <exceptionName>(e_1, ...e_n)

The first axiom describes the case where an exception is raised while the role is normally executing and is not evaluating an exception handling program. This is the case when the value `false` found in place `NoRaising`. If so, the raised exception is simply put into place `RaisedExcept` while waiting to be retrieved and solve by the corresponding CA action.

Otherwise, if place `NoRaising` contains the value `true`, the role is necessarily executing a handling program. Since a role is not allowed to raise an exception inside a handling procedure, an abort exception must be signalled. If so, instruction `Signal abort` is thus put into place `Instr` (cf. second axiom).

```
eval :: =>
   Run @, Instr (Raise (ex! en exps)),
   Ctxt c, NoRaising false ->
   Suspended @, Instr Iend,
   Ctxt c, NoRaising false,
   RaisedExcept (ex! en (subst exps c));

eval :: =>
   Run @, Instr (Raise e), NoRaising true ->
   Run @, Instr (Signal abort), NoRaising true;
```

• Instruction `Signal` <exceptionName>(e_1, ...e_n)

When a role signals an exception, it completely stops executing its program and simply ends with the signalled exception for outcome. A value `Iend` is put into both places `Instr` and `InstrSnd`, the signalled exception `e` is put in `Outcome` and a value `true` is assigned to place `NoRaising` to force an abort exception in case other roles attempt to raise an exception afterwards (cf. third axiom of method `exRaised`).

```
eval :: =>
   Run @, Instr(Signal e), InstrSnd i,
   NoRaising b ->
   Instr Iend, InstrSnd Iend,
   NoRaising true, Outcome e;


exRaised(nil) :: =>
   Run @, NoRaising false -> Suspended @, NoRaising false;

exRaised(e) :: =>
   Suspended @, RaisedExcept e, NoRaising false ->
   Suspended @, NoRaising false;

exRaised(abort) :: =>
   NoRaising true -> NoRaising true;
```

• Instruction `Assign` $< exp >$ `to` $< x >$

This instruction is used to assign the value of expression `exp` to variable `v` in the role's evaluation context `c`. The specification of contexts, assignments and substitution rules are defined in the Adt modules `Assignements`, `Environment` and `Substitution`.

```
eval :: =>
   Run @, Instr (Assign exp To x), Ctxt c ->
   Run @, Instr Iend, Ctxt (assign (subst exp c) to x within c);
```

• Instruction `Execute` $<o.m(e_1, ...e_n)>$

This instruction consists in requesting an operation `m`, with arguments `exps`, to be executed by a given object `o` appearing in the role's set of accessible objects (cf. place `Obj`). The operation is put into place `ObjMgrReq` and a token is taken out of place `Run` to suspend the role's evaluation process while waiting for the operation request to be served by the object `o`.

```
eval :: =>
   Run @, Instr (Execute (evt o m exps)),
   Obj o, Ctxt c ->
   ObjMgrReq (evt o m (subst exps c)),
   Obj o, Ctxt c;
```

2.2.3 Class `IntExtObject`

General principles

Class `IntExtObject` specifies how objects, internal or external to some (possibly nested) CA Action, ensure the preservation of ACID properties when executing operations requested by roles. The solution proposed by class `IntExtObject` suggests that each object maintains a tree of *state versions*: these state versions correspond to the different views that CA actions have of the object's state during their execution. They are kept in a tree structure according to the nesting relation of CA actions. (For example, if CA action A_2 operating on object o is nested in CA action A_1, object o will keep a tree of state versions in which the state version associated to A_2 is a child of the state version associated to A_1).

When a CA action commits, its final version of the object's state becomes the current state of the object. To ensure operation serializability (and ACID properties), this update is allowed only if the object has not meanwhile been modified by another CA action. This is verified by checking that the object's current state still corresponds to the object's state view the CA action had when it started. This condition must be satisfied before making any persistent update of the object's current state.

On the other hand, when a CA action aborts, the solution is quite simple: no state modification is persistent and no update is thus made to the object's current state.

Object state versions

Each internal/external object has a reference on a tree of state versions. State versions are represented by instances of class `objVersion`. For each CA

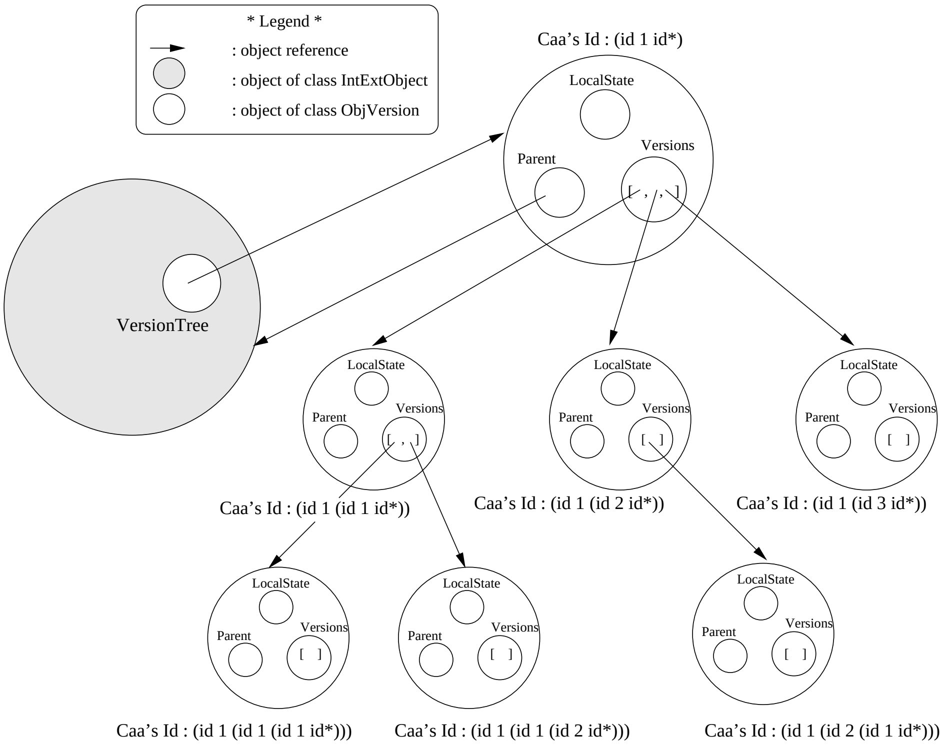

Figure 4: An internal/external object with its tree of state versions.

action operating on the object, an object state version instance is created in the tree to represent the local view which this CA action has of the object's state. State version instances are ordered in the tree according to the nesting relation which exists between the CA actions. The nesting relation can be deduced from the identity associated to each CA action when it is created. More precisely, if CA action A_2 is nested in CA action A_1, then A_1's identity is a prefix of A_2's. For example, and as shown on Figure 4, the object state version associated to a CA action having id "121" (in algebraic terms: *id(1, id(2, id(1, id*))))*) is a son of the object state version associated to the CA action having id "12"(in algebraic terms: *id(1, id(2, id*)))*).

Serving operation requests

As described by transition `serveRequest`, when an operation request is pending in the place `ObjMgrReq` of a role instance, the object concerned by the request synchronizes with the role instance (cf. `r.evtReq(evt(Self,m,args),i)`) to get two important parameters: (1) the operation (or *event*) to execute and (2) the identity of the CA action to which the requesting role belongs. The object can then apply the operation to the state version corresponding to the given CA action identity. But this first requires a search in the tree of state versions (cf. `v.apply(evt(Self,m,args), i)`).

```
serveRequest With
    (r.evtReq(evt(Self,m,args), i)
     .. v.apply(evt(Self,m,args), i)) :: =>
    VersionTree v -> VersionTree v;
```

Among others, method `apply` (taking arguments of type `event` and `caaId`) of class `objVersion` is responsible for finding the state version related to the given CA action identity `caaId` (cf. third axiom of the method). If the appropriate state version is not found, a state version instance is created and initialized (see second axiom).

Once the wanted state version is found in the tree (cf. first axiom) or has been created, appropriate algebraic values are put into place `ApplyAll` so as to allow the firing of transition `applyAll`. This last transition consists in applying the requested operation not only to the state version found but also to all its descendants in the tree (since all nested CA actions must see immediately the modifications concurrently made by their parent).

```
apply(ev, i)  ::
    i = ii
    => LocalVid ii, Versions vl ->
       LocalVid ii, Versions vl, ApplyAll ev vl;

apply(ev, i) With v.createObjVersion((prefx i ii), Self)
                .. v.apply(ev,i) ::
    (!(i=ii)) & (find(prefx(i,ii),vl) = vers(id*))
    => LocalVid ii, Versions vl ->
       LocalVid ii, Versions [vers(prefx(i,ii),v) |  vl] ;

apply(ev, i) With v.apply(ev,i) ::
    (!(i=ii)) & (find(prefx(i,ii),vl) = vers(i,v))
    => LocalVid ii, Versions vl ->
       LocalVid ii, Versions vl;


applyAll :: =>
    ApplyAll ev [] -> ToExecute ev ;
applyAll With v.startApplyAll(ev) :: =>
    ApplyAll ev [vers(i,v) | vl] -> ApplyAll ev vl ;


startApplyAll(ev) :: Versions vl -> Versions vl, ApplyAll ev vl
```

Each object state version to which the operation is applied thus ends with an event in its place `ToExecute`. This enables transition `execute` to fire and to modify the local state of the state version according to the **specific** behavior of the object for that operation. Being generic, class **objVersion** can't give the rules describing the specific execution of objects' operations (events). For each type of object (be it an account, a store, a book or a buffer object, etc.), transition `execute` must be explicitely specified in a subclass (inheriting from class `objVersion`) describing the particular behavior of the object for each possible operation it can execute. Related axioms should have the following shape:

```
execute :: => ToExecute (evt o m1 args), LocalState s1 -> LocalState s2 ;
```

As for `commit` and `abort` operations, since they are common to all object state versions, their respective executions are described directly in the generic class `objVersion`, by two axioms related to transition `execute`. In case of a `commit`, the object state version induces the update of its parent's current state, according to its own current state. In case of an `abort`, nothing happens apart from changing the value in place `Ended` from `false` to `true`.

```
execute With p.localState(s) .. p.update(ss) ::
    eqList(s, sss) = true =>
    Parent p, LocalState ss, InitState sss,
    Ended false, ToExecute (evt o commit args) ->
    LocalState ss, InitState sss, Ended true ;

execute :: =>
    ToExecute (evt o abort args), Ended false -> Ended true;
```

3 Conclusions

Following the first article written on COALA ([VBB+99]), this report was meant to give a precise and detailed explanation of the semantics of COALA written in CO-OPN/2. This semantic work led us to achieve many aims:

- Clarification of the fundamental concepts related to CA actions. Formal and clear definition of CA actions' properties and principles.
- Production of a formal document which can be used as a basic reference for future implementations of COALA compilers and run-time systems for CA actions.
- Providing COALA with new testing, refinement and validation tools: indeed, tools already developped for CO-OPN/2 can be applied to the CO-OPN/2 semantic definitions of COALA.
- Interesting development experience of a significant size CO-OPN/2 specification; this contributed to pointing out some of CO-OPN/2's deficiencies and expressivity weaknesses.
- Use of CO-OPN/2's checking tools to verify both the syntax and the types of the CO-OPN/2 specifications defining COALA's semantics.

Further works concerning COALA and CO-OPN/2 are in progress. A Java prototyping tool for CO-OPN/2 specifications is being developped. It is hoped that with these forthcomming tools, a simple Java prototype of a COALA compiler with a CA action runtime can soon be obtained.

References

[BB97] M. Buffo and D. Buchs. A coordination model for distributed object systems. In *Proceedings of the Second International Conference on Coordination Models and Languages COORDINATION'97, September 1997*, volume 1282 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[BBG97] O. Biberstein, D. Buchs, and N. Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In *DeVa Second Year Report Delivrables*, pages 103–168. Esprit Long Term Research Project 20072 - DeVa, December 1997.

[BBP96] S. Barbey, D. Buchs, and C. Péraire. A theory of specification-based testing for object-oriented software. In *Proceedings of EDCC2 (European Dependable Computing Conference), Taormina, Italy, October 1996*, LNCS (Lecture Notes in Computer Science) 1150, pages 303–320. Springer-Verlag, 1996. Also available as Technical Report (EPFL-DI No 96/163), Published in DeVa first year report (December 96).

[Bib97] O. Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems.* PhD thesis, University of Geneva, 1997. Thesis No 2919.

[GR93] J.N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publ., 1993.

[Gro96] Object Management Group. Object transaction service. Draft 4, OMG document, November 1996.

[PBB98] C. Péraire, S. Barbey, and D. Buchs. Test selection for object-oriented software based on formal specifications. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98), Shelter Island, New York, USA, June 1998*, pages 385–403. Chapman & Hall, 1998. Also available as Technical Report (EPFL-DI No 97/252), Published in DeVa second year report (January 98).

[Pér98] Cécile Péraire. *Formal Testing of Object-Oriented Software: from the Method to the Tool.* PhD thesis, no 1904, Ecole Polytechnique Fédérale de Lausanne, Département d'Informatique, 1015 Lausanne, Switzerland, 1998.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Trans. Soft. Eng.*, SE-1(2):220–232, 1975.

[RRS+97a] B. Randell, A. Romanovsky, R. Stroud, J. Xu, and F. Zorzo. Coordinated atomic actions: from concept to implementation. Technical Report 595, Computing Dept., University of Newcastle upon Tyne, 1997.

[RRS+97b] A. Romanovsky, B. Randell, R. Stroud, J. Xu, and A. Zorzo. Implementation of blocking coordinated atomic actions based on forward error recovery. *Journal of System Architecture (Special Issue on Dependable Parallel Computing Systems)*, 43(10):687–99, 1997.

[Ser99] Giovanna Di Marzo Serugendo. *Stepwise Refinement of Formal Specifications Based on Logical Formulae: from COOPN/2 Specifications to Java Programs.* PhD thesis, no 1931, Ecole Polytechnique Fédérale de Lausanne, Département d'informatique, CH-1015, Lausanne, Suisse, 1999.

[VBB+99] J. Vachon, D. Buchs, M. Buffo, G. DiMarzo Serugendo, B. Randell, S. Romanovsky, R. Stroud, and J. Xu. Coala - a formal language for coordinated atomic actions. In *DeVa Third Year Report, Deliverables:Part2 (Papers)*. Esprit Long Term Research Project 20072 - DeVa, December 1999.

[XRR+95] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proc. 25th Int. Symp. on Fault-Tolerant Computing*, pages 499–508, Pasadena, June 1995.

[XRR98] J. Xu, A. Romanovsky, and B. Randell. Coordinated exception handling in distributed object systems: from model to system implementation. In *Proc. of the Int. Conference on Distributed Computing Systems, ICDCS-18*, Amsterdam, May 1998.

Appendix A: *The BNF Grammar of COALA*

The BNF grammar of COALA follows below. In this grammar, *name* refers to some valid identifier. As for the generators *coopnModule*, *expression*, *condition* and *type*, they correspond respectively to the generators *modules*, *expressions*, *condition* and *types* found in the grammar of CO-OPN/2 (which is not given here).

program ::= (*caaModule* | *coopnModule*) *
caaModule ::= Caa *name* ‘;’ [*caaInterface*] [*caaBody*] End *name* ‘;’
caaInterface ::= Interface [*useSection*] [*roleSection*] [*exceptionSection*]
[*precondSection*] [*postcondSection*]
caaBody ::= Body [*useSection*] [*useCaaSection*] [*objectSection*]
[*handlerSection*] [*exceptionSection*]
[*resolutionSection*] [*whereSection*] (*role*)*
useSection ::= Use (*nameList* ‘;’)+
useCaaSection ::= Use Caa (*nameList* ‘;’)+
roleSection ::= Role (*nameList* [‘:’ *typeList*] ‘;’)+
objectSection ::= Object (*nameList* ‘:’ *type* ‘;’)+
handlerSection ::= Handler (*nameList* [‘:’ *typeList*] ‘;’)+
exceptionSection ::= Exception (*nameList* [‘:’ *typeList*] ‘;’)+
precondSection ::= Precondition *expression* ‘;’
postcondSection ::= Postcondition [*expression* ‘;’] (*nameList* ‘:’ *expression* ‘;’)+
resolutionSection ::= Resolution (*excList* ‘->’ *name* [‘(’ *expressionList* ‘)’] ‘; ’)+
excList ::= *name* [‘(’ *nameList* ‘)’] (‘,’ *name* [‘(’ *nameList* ‘)’])*
whereSection ::= Where (*nameList* ‘:’ *type* ‘;’)+
role ::= Role *name* [‘(’ *nameList* ‘)’] ‘;’
instructionBlock [*whereSection*] (*handler*)* End *name* ‘;’
handler ::= Handler *name* [‘(’ *nameList* ‘)’] ‘;’
instructionBlock [*whereSection*] End *name* ‘;’
instructionBlock ::= Begin *instruction* (‘;’ *instruction*)* End
instruction ::= *empty* | *assign* | *execute*| *if* | *raise* | *signal* | *callRole*
empty ::=
assign ::= Assign *expression* To *name*
execute ::= Execute *expression*
if ::= If *condition* Then *instructionBlock*
[Else *instructionBlock*]
raise ::= Raise *name* [‘(’ *expressionList* ‘)’]
signal ::= Signal *name* [‘(’ *expressionList* ‘)’]
callRole ::= Call *name* [‘(’ *expressionList* ‘)’] Of *name*
nameList ::= *name* (‘,’ *name*)*
expressionList ::= *expression* (‘,’ *expression*)*
typeList ::= *type* (‘,’ *type*)*

Appendix B: *The CO-OPN/2 specification of COALA's semantics*

All the specifications were verified (syntax and types) and formatted using CO-OPN/2's checking tools.

B.1 Specifications of Abstract Data Types

```
;;==============
Adt Assignment;
;;==============

Interface

  Use Name, Expression, Bools;

  Sorts assignment;

  Generators
     _ / _ : varName expr -> assignment;

  Operations
     eqAssign _ _ : assignment assignment -> bool;

Body

  Axioms

    v1 = v2 => eqAssign (v1/e1) (v2/e2) = eqExpr e1 e2 ;
    !(v1 = v2) => eqAssign (v1/e1) (v2/e2) = false;

    Where
        v1, v2 : varName;
        e1, e2 : expr;

End Assignment;
```

```
;;=========
Adt Bools;
;;=========

Interface
  Use Name;
  Sort bool, varbool;
  Subsorts varbool -> bool;
           varbool -> varName;
  Generators
    true   : -> bool;
    false  : -> bool;

    varBool1 : -> varbool;
    varBool2 : -> varbool;
    varBool3 : -> varbool;
    ;; etc.
```

```
  Operations
    not   _    : bool -> bool;
    _ and _    : bool bool -> bool;
    _ or  _    : bool bool -> bool;
    _ xor _    : bool bool -> bool;
    eqBool _ _ : bool bool -> bool;
Body
  Axioms
    not true   = false;
    not false  = true;

    true and b  = b;
    false and b = false;
    b1 and b2   = b2 and b1;

    true or b  = true;
    false or b = b;
    b1 or b2 = b2 or b1;

    true xor b   = not b;
    false xor true  = false;
    b1 xor b2 = b2 xor b1;

    eqBool b b   = true;
    eqBool true false  = false;
    eqBool b1 b2 = eqBool b2 b1;

  Where
    b, b1, b2 : bool;
End Bools;


;;===============
Adt CaaIdentity;
;;===============
Interface

  Use Naturals;

  Sort caaId;

  Generators
        id* : -> caaId;
        id _ _: natural caaId  -> caaId;

  Operations

        prefx _ _ : caaId caaId -> caaId;
Body
  Axioms
```

```
        !(x=y) => prefx(id(x,xid), id*) = id(x,id*);
        prefx (id(x,xid), id(x,yid)) = id(x,prefx(xid,yid));

        Where
          x, y : natural;
          xid, yid : caaId;

End CaaIdentity;


;;=====================================
Adt Environment As List(Assignment);
;;=====================================

  Morphism
    elem -> assignment;
    eqElem -> eqAssign;
  Rename
    list -> environ;

Interface
  Use ListOfExpr, Bools;
  Operations
     getValue _ _ : varName environ -> expr;
     assign _ to _ within _ : expr varName environ -> environ;

Body

  Axioms

    getValue x [] = undef;
    x = y    =>  getValue x  (y/v 'c) = v ;
    !(x = y) =>  getValue x  (y/v 'c) = getValue x c ;

    assign v to x within [] = [] ;
    x = y  =>
      assign vx to x within (y/vy 'c) = (y/vx 'c);
    !(x = y)  =>
      assign vx to x within (y/vy 'c) = y/vy ' (assign vx to x within c);

  Where
    x, y : varName;
    v, vx, vy : expr;
    c : environ;

End Environment;
```

```
;;==========
Adt Event ;
;;==========

Interface

  Use IntExtObject, Name, ListOfExpr;

  Sort event;

  Generators
    evt _ _ _ : objType methName exprlist -> event;

End Event;

;;===============
Adt Exception ;
;;===============

Interface

  Use Name, ListOfExpr, Bools;

  Sorts exception;

  Generators
     ex! _ _ : exceptName exprlist -> exception;
     nil : -> exception;
     fail : -> exception;
     abort : -> exception;

  Operations
     eqExcept _ _ : exception exception -> bool;

Body

  Axioms
   eqExcept x x = true;
   eqExcept x y = eqExcept y x ;
   e1 = e2 =>
        eqExcept (ex! e1 exps1) (ex! e2 exps2) = eqList exps1 exps2;
   !(e1 = e2) =>
        eqExcept (ex! e1 exps1) (ex! e2 exps2) = false;
   eqExcept abort nil = false;
   eqExcept fail nil = false;
   eqExcept fail abort = false;
   eqExcept (ex! e exps) nil = false;
   eqExcept (ex! e exps) abort = false;
   eqExcept (ex! e exps) fail = false;


  Where
    x, y : exception;
    e, e1, e2 : exceptName;
    exps, exps1, exps2 : exprlist;

End Exception;
```

```
;;==============
Adt Expression;
;;==============

Interface
  ;; Add here all ADT modules used in the roles' programs.
  ;; Complete the subsort relation accordingly.

  Use Naturals, Bools;                        ;; *to be completed*

  Sorts expr;

  Subsort     natural -> expr;
              bool -> expr;                   ;; *to be completed*

  Generators
    undef : -> expr;
    eqExpr _ _ : expr expr -> bool;

;;  ** To be completed for each sort. **
    exprToBool _ : expr -> bool;
    exprToNatural _ : expr -> natural;

Body

  Axioms

   eqExpr b1 b2 = eqBool b1 b2 ;
   eqExpr n1 n2 = eqNat  n1 n2 ;

;;  ** To be completed for each sort. **
   exprToBool b = b;
   exprToNatural n = n;

   Where
     b, b1, b2 : bool;
     n, n1, n2 : natural;

End Expression;


;;===============
Adt Instruction;
;;===============

Interface

  Use Name, Expression, Exception, Event, ListOfExpr;
  Use Role, IntExtObject, ListOfObjects;
  Use Bools;

  Sort instruction;
  Sort catchInst;

  Subsort instruction -> catchInst;

  Generators
```

```
    Execute _ : event -> instruction;
    CallRole _ withArgs _ :
        roleType objlist -> instruction;
    Sequence _ _ : instruction instruction -> instruction;
    If _ Then _ Else _ :
        bool instruction instruction -> instruction;
    Raise _ : exception -> instruction;
    Signal _ : exception -> instruction;
    Assign _ To _ : expr varName -> instruction;
    Iend : -> instruction;
```

End Instruction;

;;========
Adt Name;
;;========

Interface

```
  Sorts        name, methName, placeName,
               exceptName, varName;
  Subsorts     methName -> name;
               placeName -> name;
               exceptName -> name;
               varName -> name;
```

Generators

;; Add generators for exception names

;; Add generators for method names

```
        commit : -> methName;
        abort  : -> methName;
```

;; Add generators for place names

;; (Generators of variable names for a given sort are
;; defined in the Adt module defining this sort.)

End Name;

```
;;============
Adt Naturals;
;;============

Interface

  Use  Bools, Name;

  Sort natural, varnat;

  Subsort varnat -> natural;
          varnat -> varName;

  Generators
    0       : -> natural;
    succ _  : natural -> natural;
    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20 : -> natural;

    varNat1 : -> varnat;
    varNat2 : -> varnat;
    varNat3 : -> varnat;
    ;; etc.

  Operations
    _ + _     : natural natural -> natural;
    _ - _     : natural natural -> natural;
    _ * _     : natural natural -> natural;
    _ / _     : natural natural -> natural;
    _ % _     : natural natural -> natural;
    eqNat _ _ : natural natural -> bool;
    _ <= _    : natural natural -> bool;
    _ < _     : natural natural -> bool;
    _ > _     : natural natural -> bool;
    _ >= _    : natural natural -> bool;
    max _ _   : natural natural -> natural;
    min _ _   : natural natural -> natural;
    even _    : natural -> bool;
    2** _     : natural -> natural;
    _ ** 2    : natural -> natural;

Body
  Axioms

    0 + n = n;
    (succ n1) + n2 = succ (n1 + n2);
    n1 + n2 = n2 + n1;

    0 - n = 0;
    (succ n) - 0 = succ n ;
    (succ n1) - (succ n2) = (n1 - n2);

    0 * n = 0;
    (succ n1) * n2 = (n1 * n2) + n2;

    0/(succ n) = 0;
```

```
    (n1 < n2) = true =>
        (succ n1)/(succ n2) = 0;
    (n1 >= n2) = true =>
        (succ n1)/(succ n2) = succ (((succ n1) - (succ n2)) / (succ n2)) ;

    n1 % (succ n2) = n1 - ( (n1/(succ n2)) * (succ n2));

    eqNat 0 0 = true;
    eqNat 0 (succ n) = false;
    eqNat (succ n1) (succ n2) = eqNat n1 n2;
    eqNat n1 n2 = eqNat n2 n1;

    n < 0 = false;
    0 < (succ n) = true;
    (succ n1) < (succ n2) = n1 < n2;

    0 > n = false;
    (succ n) > 0 = true;
    (succ n1) > (succ n2) = n1 > n2;

    n1 >= n2 = (n1 > n2) or (eqNat n1 n2);
    n1 <= n2 = (n1 < n2) or (eqNat n1 n2);

    even 0 = true;
    even (succ n) = not even n;

    2**0 = (succ 0);
    2**(succ n) = 2 * (2**n);

    (n1 >= n2) = true => max n1 n2 = n1;
    max n1 n2 = max n2 n1;

    (n1 >= n2) = true => min n1 n2 = n2;
    min n1 n2 = min n2 n1;

    1  = succ 0;   2  = succ 1;   3  = succ 2;   4  = succ 3;
    5  = succ 4;   6  = succ 5;   7  = succ 6;   8  = succ 7;
    9  = succ 8;   10 = succ 9;   11 = succ 10;  12 = succ 11;
    13 = succ 12;  14 = succ 13;  15 = succ 14;  16 = succ 15;
    17 = succ 16;  18 = succ 17;  19 = succ 18;  20 = succ 19;
  Where
    n, n1, n2 : natural;
End Naturals;

;;==============================
Adt ObjState As List(Stplace);
;;==============================
Morphism
    elem -> stPlace ;
Rename
    list -> objState ;

End ObjState;
```

```
;;===========
Adt Outcome;
;;===========

Interface

  Use Exception, Bools;

  Sort outcome;

  Subsorts exception -> outcome;

  Generators
    normal : -> outcome;

  Operations
    eqOutc _ _ : outcome outcome -> bool;
    result _ _ : outcome outcome -> outcome;

Body

  Axioms

    eqOutc outc outc = true;
    eqOutc outc1 outc2  = eqOutc outc2 outc1;

    eqOutc e1 e2 = eqExcept e1 e2;
    eqOutc normal e = false;

    result outc fail = fail;
    result fail outc = fail;
    eqOutc outc fail = false => result outc abort = abort;
    eqOutc outc fail = false => result abort outc = abort;
    eqOutc outc1 outc2 = false => result outc1 outc2 = abort;
    eqOutc outc1 outc2 = true => result outc1 outc2 = outc1;

    Where
      e, e1, e2 : exception;
      outc, outc1, outc2 : outcome;

End Outcome;

;;===============
Adt SolveExcept;
;;===============

Interface

  Use ListOfExcept, CaaIdentity;

  Operations
     solve _ _ : caaId exceptlist -> exception;

Body

  Axioms

     fail isIn? exlist = true =>
        solve i exlist = fail;
```

```
     (not(fail isIn? exlist)) and
     (abort isIn? exlist) = true =>
        solve i exlist = abort;
```

;; Axioms are to be added to represent the resolution tree
;; of each individual CA action.

```
     Where
        i : caaId;
        exlist : exceptlist;

End SolveExcept;


;;===========
Adt Stplace;
;;===========

Interface

  Use Name, ListOfExpr, Bools;

  Sort stPlace;

  Generators
   pl _ _ : placeName exprlist -> stPlace;

  Operations
   eqPlace _ _ : stPlace stPlace -> bool;

Body

  Axioms

   n1 = n2 => eqPlace (pl n1 e1) (pl n2 e2) = eqSet e1 e2;
   !(n1 = n2) => eqPlace (pl n1 e1) (pl n2 e2) = false;

   Where
     n1, n2 : placeName;
     e1, e2 : exprlist;

End Stplace;


;;=================
Adt Substitution;
;;=================
```

;; Substitution rules must be defined recursively over the structure
;; of all expression terms.

```
Interface

  Use Environment, Expression, ListOfExpr;

  Operations
     subst _ _ : exprlist environ -> exprlist;
```

```
subst _ _ : expr environ -> expr;
```

Body

Axioms

```
subst [] c = [];
subst (e'es) c = (subst e c) ' (subst es c);
```

;;
;; Substitution rules for Bools
;;
subst bvar c = getValue bvar c;

```
subst true c = true;
subst false c = false;
subst (not b) c = not (exprToBool (subst b c));
subst (b1 and b2) c =
    (exprToBool (subst b1 c)) and (exprToBool (subst b2 c));
subst (b1 or b2) c =
    (exprToBool (subst b1 c)) or (exprToBool (subst b2 c));
subst (b1 xor b2) c =
    (exprToBool (subst b1 c)) xor (exprToBool (subst b2 c));
subst (eqBool b1 b2) c =
    eqBool (exprToBool (subst b1 c)) (exprToBool (subst b2 c));
```

;;
;; Substitution rules for Naturals
;;

```
subst (1) c = 1; subst (2) c = 2; subst (3) c = 3;
subst (4) c = 4; subst (5) c = 5; subst (6) c = 6;
subst (7) c = 7; subst (8) c = 8; subst (9) c = 9;

subst nvar c = getValue nvar c;
subst (0) c = 0;
subst (succ n) c = succ (exprToNatural (subst n c));
subst (n1 + n2) c =
    (exprToNatural (subst n1 c)) + (exprToNatural (subst n2 c));
subst (n1 - n2) c =
    (exprToNatural (subst n1 c)) - (exprToNatural (subst n2 c));
subst (n1 * n2) c =
    (exprToNatural (subst n1 c)) * (exprToNatural (subst n2 c));
subst (n1 / n2) c =
    (exprToNatural (subst n1 c)) / (exprToNatural (subst n2 c));
subst (n1 % n2) c =
    (exprToNatural (subst n1 c)) % (exprToNatural (subst n2 c));
subst (eqNat n1 n2) c =
    eqNat (exprToNatural (subst n1 c)) (exprToNatural (subst n2 c));
subst (n1 <= n2) c =
    (exprToNatural (subst n1 c)) <= (exprToNatural (subst n2 c));
subst (n1 < n2) c =
    (exprToNatural (subst n1 c)) < (exprToNatural (subst n2 c));
subst (n1 > n2) c =
```

```
        (exprToNatural (subst n1 c)) > (exprToNatural (subst n2 c));
    subst (n1 >= n2) c =
        (exprToNatural (subst n1 c)) >= (exprToNatural (subst n2 c));
    subst (max n1 n2) c =
        max (exprToNatural (subst n1 c)) (exprToNatural (subst n2 c));
    subst (even n) c =
        even (exprToNatural (subst n c));
    subst (2** n) c =
        2** (exprToNatural (subst n c));
```

Where

```
      bvar: varbool;
      nvar: varnat;
      b, b1, b2 : bool;
      c : environ;
      e : expr;
      es : exprlist;
      n, n1, n2 : natural;
```

End `Substitution;`

;;=========
Adt `Token;`
;;=========

Interface

Sort `token;`

Generator
```
    @ : -> token;
```

End `Token;`

;;=============================
Parameter Adt `ComparableElem ;`
;;=============================

Interface

Use `Bools;`

Sort `elem;`

Operation
```
    eqElem _ _ : elem elem -> bool;
```

Body

Theorems
;; usual equivalence relation properties
;; reflexivity

```
    eqElem x x = true;

    ;; symmetry
    eqElem x y = true => eqElem y x = true;

    ;; transitivity
    (eqElem x y = true) & (eqElem y z = true) => eqElem x z = true;

    Where
      x, y, z : elem;

End ComparableElem;


;;=================================
Generic Adt List(ComparableElem);
;;=================================

Interface

  Use  Naturals, Bools, ComparableElem;

  Sort list;

  Generators
    []     : -> list;
    _ ' _ : elem  list -> list;

  Operations
    _ | _          : list list -> list;          ;; concatenation
    # _            : list -> natural;            ;; number of compo-
nents
    ztake _ from _ : natural list -> list;       ;; first n component(s)
    drop _ from _ : natural list -> list;        ;; l - ztake(n,l)
    head _         : list -> elem;               ;; first component
    tail _         : list -> list;               ;; l - first component
    empty? _       : list -> bool;
    reverse _      : list -> list;
    eqList _ _     : list list -> bool;
    eqSet _ _      : list list -> bool;

    remove _ _     : elem list -> list;          ;; remove an element
    _ isIn? _      : elem list -> bool;          ;; check if elem in the list
    _ ~ _          : list list -> bool;          ;; lists have same elements
    dropUntil _ _ : elem list -> list;           ;; drop elements until
                                                 ;; a given elem is found

Body

  Axioms
    ([] | l1)  = l1;
    ((e ' l1) | l2) = (e ' (l1 | l2));

    #([]) = 0;
    #(e ' l1) = succ(#(l1));

    ztake n from []       = [];
```

```
ztake 0 from e ' l1    = [];
ztake succ(n) from e ' l1 = e ' ztake n from l1;

drop n from []       = [];
drop 0 from (e'l1)    = (e'l1);
drop succ(n) from e ' l1 = (drop n from l1);

head(e'l1) = e;

tail([]) = [];      ;; if is-empty(l1) then tail(l1) = []
tail(e'l1) = l1;

empty?([]) = true;
empty?(e'l1) = false;

reverse([]) = [];
reverse(e'l1) = (reverse l1) | (e'[]);

eqList [] [] = true;
eqList (e'l1) [] = false;
eqList [] (e'l1) = false;
eqList (e1'l1) (e2'l2) = (eqElem e1 e2) and (eqList l1 l2);

eqSet [] [] = true;
eqSet (e'l1) [] = false;
eqSet [] (e'l1) = false;
eqSet (e1'l1) l2 = (e1 isIn? l2) and eqSet l1 (remove e1 l2);

remove e [] = [];
(eqElem e e1) = true  => remove e (e1'l1) = l1;
(eqElem e e1) = false => remove e (e1'l1) = e1 ' (remove e l1);

e isIn? [] = false;
e isIn? (e1'l1) = (eqElem e e1) or (e isIn? l1);

[] ~ [] = true;
(e1'l1) ~ l =  (e1 isIn? l) and (l1 ~ (remove e1 l));

dropUntil e [] = [];
(eqElem e e1) = true => dropUntil e (e1 ' l1) = l1;
(eqElem e e1) = false => dropUntil e (e1 ' l1) = dropUntil e l1;
```

Theorem

```
(ztake n from l ) | (drop n from l) = l;

reverse(reverse(l)) = l;

;; usual equivalence relation properties
eqList l l = true;                              ;; reflexivity

eqList l1 l2 = true => eqList l2 l1 = true;   ;; symmetry

;; transitivity
(eqList l1 l2 = true) & (eqList l2 l3 = true) => eqList l1 l3 = true;
```

Where

```
l, l1, l2, l3 : list;
```

```
    e, e1, e2     : elem;
    n : natural;

End List;


;;===================================================
Adt ListOfExcept As List(Exception);
;;===================================================

Morphism
    elem -> exception ;
    eqElem -> eqExcept;
Rename
    list -> exceptlist ;

End ListOfExcept;


;;==================================================
Adt ListOfExpr As List(Expression);
;;==================================================

  Morphism
    elem -> expr;
    eqElem -> eqExpr;
  Rename
    list -> exprlist;
End ListOfExpr;


;;=========================
Adt ListOfObjects;
;;=========================

Interface

  Use IntExtObject;
  Sort objlist;

  Generators

    [] : objlist;
    _ ' _ : objType objlist -> objlist;

End ListOfObjects;
```

```
;;================
Adt ListOfRoles;
;;================

Interface

  Use Role, Bools;
  Sort rolelist;

  Generators

    [] : rolelist;
    _ ' _ : roleType rolelist -> rolelist;

  Operations

    _ isIn? _ : roleType rolelist -> bool;

    remove _ _ : roleType rolelist -> rolelist;

Body

  Axioms

    r isIn? [] = false;
    r1=r2 => r1 isIn? (r2'rlist) = true;
    !(r1=r2) => r1 isIn? (r2'rlist) = r1 isIn? rlist;

    remove r [] = [];
    r1 = r2  => remove r1 (r2'rlist) = rlist;
    !(r1 = r2)  => remove r1 (r2'rlist) = r2 ' (remove r1 rlist);

    Where
      r, r1, r2 : roleType;
      rlist : rolelist;

End ListOfRoles;


;;================
Adt ListOfVersions;
;;================

Interface

  Use CaaIdentity, ObjVersion ;

  Sorts version, versionList ;

  Generators
        vers _ : caaId -> version;
        vers _ _ : caaId objVersionType -> version;
        [] : versionList;
        [_|_] : version versionList -> versionList;

  Operations

        find _ _ : caaId versionList -> version;
```

Body

Axioms

```
find(i,[]) = vers(id*);
find(i,[vers(i,ov)|vl]) = vers(i,ov);
!(i=i') =>  find(i,[vers(i',ov)|vl]) = find(i,vl);
```

Where

```
i, i' : caaId;
ov : objVersionType;
vl : versionList;
```

End `ListOfVersions;`

B.2 Specification of class Caa

```
;;==========
Class Caa ;
;;==========

Interface

  Use CaaIdentity, ListOfObjects, ListOfRoles;
  Type caaType;

  Creation
    createCaa _ _ _ : caaId rolelist objlist;

Body

  Use Token, Exception, ListOfExcept,
      Outcome, SolveExcept;

  Transitions
    acceptTest;
    endRoles;
    getExcept;
    handleExcept;
    startRoles;

  Places
    CalledRoles _ _ : rolelist rolelist;
    ExceptRaised _ : exceptlist;
    Id _ : caaId;
    Obj _ : objlist;
    Outcome _ : outcome;
    Roles _ : rolelist;

    AcceptTest _ : token;
    CAAFinished _ : token;
    EndR _ : token;
    Run _ : token;
    StartR _ : token;
    Suspended-Get _ : token;
    Suspended-Handle _ : token;

  Axioms

;;            ---------------------------------------------
;;                 CA ACTION'S CREATION
;;            ---------------------------------------------

    createCaa(i, rlist, caaObj) :: =>
        -> Id i, Roles rlist, Obj caaObj;

;;            --------------------------------------------------
;;               CA ACTION'S MAIN EXECUTION:
;;              general coordination of roles
;;            --------------------------------------------------
```

```
    startRoles :: =>
        StartR @, Roles [], CalledRoles rcalled rcalling ->
        Run @, Roles rcalled, CalledRoles rcalled rcalling;

    startRoles With (c.callRole(r, extObj)
                   .. r.start(extObj, caaObj)) :: =>
        StartR @, Roles (r ' rlist), Obj caaObj,
        CalledRoles rcalled rcalling ->
        StartR @, Roles rlist, Obj caaObj,
        CalledRoles (r ' rcalled) (c ' rcalling);



    acceptTest With r.outcome(outc) :: =>
        Run @, CalledRoles (r ' rlist) clist ->
        AcceptTest @, CalledRoles rlist clist, Outcome outc;

    acceptTest :: =>
        AcceptTest @, CalledRoles [] clist, Roles rlist ->
        EndR @, CalledRoles rlist clist, Roles rlist;

    acceptTest With r.outcome(outc1) :: =>
        AcceptTest @, CalledRoles (r ' rlist) clist,
        Outcome outc2 ->
        AcceptTest @, CalledRoles rlist clist,
        Outcome (result(outc1, outc2));

    endRoles :: =>
        EndR @, CalledRoles [] [] -> CAAFinished @;

    endRoles With  r.abort .. c.signalOutcome(abort) :: =>
        EndR @, Outcome abort, CalledRoles (r ' rcalled) (c ' rcalling) ->
        EndR @, Outcome abort, CalledRoles rcalled rcalling ;

    endRoles With r.commit .. c.signalOutcome(outc) ::
        not(eqOutc(outc, abort)) = true =>
        EndR @, Outcome outc, CalledRoles (r ' rcalled) (c ' rcalling) ->
        EndR @, Outcome outc, CalledRoles rcalled rcalling ;

;;        -------------------------------------------------------------------
;;          SOLVING AND HANDLING RAISED EXCEPTIONS
;;        -------------------------------------------------------------------

    getExcept With r.exRaised(e) ::
        (not(eqExcept(e, nil))) and (r isIn? rlist) = true =>
        AcceptTest @, CalledRoles rlist clist,
        Roles orlist, Outcome outc ->
        Suspended-Get @, CalledRoles (remove r orlist)  clist,
        Roles orlist;

    getExcept With r.exRaised(e) ::
        (not(eqExcept(e, nil))) and (r isIn? rlist) = true =>
        Run @, CalledRoles rlist clist ->
        Suspended-Get @, CalledRoles (remove r rlist)  clist;
```

```
getExcept With r.exRaised(e) :: =>
    Suspended-Get @, CalledRoles (r ' rlist) clist,
    ExceptRaised exclist ->
    Suspended-Get @, CalledRoles rlist clist,
    ExceptRaised (e ' exclist);

getExcept :: =>
    Suspended-Get @, CalledRoles [] clist,
    Roles rlist, ExceptRaised exclist ->
    Suspended-Handle @, CalledRoles rlist clist,
    Roles rlist, ExceptRaised ((solve(i,exclist))'[]);

handleExcept :: =>
    Suspended-Handle @, CalledRoles [] clist,
    Roles rlist, ExceptRaised (e'[]) ->
    Run @, CalledRoles rlist clist,
    Roles rlist, ExceptRaised ([]);

handleExcept With r.handle(e) :: =>
    Suspended-Handle @, CalledRoles (r ' rlist) clist,
    ExceptRaised (e'[]) ->
    Suspended-Handle @, CalledRoles rlist clist,
    ExceptRaised (e'[]);

Where

    c : roleType;
    e : exception;
    exclist : exceptlist;
    extObj, caaObj: objlist;
    i : caaId;
    outc, outc1, outc2 : outcome;
    r : roleType;
    rlist, orlist, clist: rolelist;
    rcalled, rcalling : rolelist;

End Caa;
```

B.3 Specification of class `Role`

```
;;==========
Class Role;
;;==========
Interface

  Use ListOfObjects, CaaIdentity, Event, Exception,
      Environment, Instruction, Outcome;

  Type roleType;

  Methods

     abort;
     commit;

     callRole _ _ : roleType objlist;
     evtReq _ _ : event caaId;
     exRaised _ : exception;
     handle _ : exception;
     outcome _ : outcome;
     start _ _ : objlist objlist;
     signalOutcome _ : outcome;

  Creation
    createRole _ _ _: environ instruction caaId;


Body

  Use Substitution, IntExtObject, Token,
      Bools, Name, Expression, ListOfExpr ;

  Transitions
     eval;
     execCommit;
     execAbort;
     putObj;


  Places
     CaaIdentity _ : caaId;
     Ctxt _ : environ;          ;; local variables
     ExecCommit _ : token;
     ExecAbort _ : token;
     Instr _ : instruction;
     InstrSnd _ : instruction;
     ObjMgrReq _ : event;
     Outcome _ : outcome;
     PutObj _ : objlist;
     RoleToCall _ _: roleType objlist;
     RaisedExcept _ : exception;
```

```
    Obj _ : objType;          ;; internal and external objects

    NoRaising _ : bool;
    Run _ : token;
    Suspended _ : token;
```

Axioms

```
;;          -----------------------------------------------------
;;              CREATION AND START OF A ROLE
;;          -----------------------------------------------------

    createRole(c, i, id) :: =>
       -> Instr i,                       ;; The role's program.
          Ctxt c,                        ;; The role's local variables.
          CaaIdentity id;                ;; The id of the Caa the roles
                                         ;; belongs to.

    start(oext, ocaa) :: =>
       -> Run @, NoRaising false,
          PutObj oext, PutObj ocaa;

    putObj :: => PutObj [] -> ;
    putObj :: => PutObj (o ' ol) -> PutObj ol, Obj o ;

;;                 -------------------------------------------
;;                  CALLING ANOTHER ROLE
;;                 -------------------------------------------

;; Calling another role
;; ------------------------------

    callRole(r, args) :: => RoleToCall r args -> ;

;; Receiving the outcome of another role
;; ---------------------------------------------

    signalOutcome(normal) :: => -> Instr Iend, Run @;

    signalOutcome (e) ::
       !(e = normal) => -> Instr (Raise e), Run @;

    outcome(outc) :: => Outcome outc -> Outcome outc;

;;             -------------------------------------------------------
;;              EXCEPTION RAISING AND HANDLING
;;             -------------------------------------------------------

;; Collection of concurrently raised exceptions
;; --------------------------------------------------------

    exRaised(nil) :: =>
       Run @, NoRaising false -> Suspended @, NoRaising false;

    exRaised(e) :: =>
       Suspended @, RaisedExcept e, NoRaising false ->
```

```
        Suspended @, NoRaising false;

     exRaised(abort) :: =>
        NoRaising true -> NoRaising true;


;; Calling handler for exception recovery
;; ---------------------------------------------
;;   **      Method "handle" must be specified in the      **
;;   **              subclass specification.               **

;;   For each exception, the appropriate handling
;;   program must be put in place Instr to be executed.

;;    handle(totoExcept) :: =¿
;;      Suspended @, NoRaising false, Instr i1, InstrSnd i2 -¿
;;      Instr totoblabla, InstrSnd Iend, Run @, NoRaising true;

;;    handle(titiExcept) :: =¿
;;      Suspended @, NoRaising false, Instr i1, InstrSnd i2 -¿
;;      Instr titiblabla, InstrSnd Iend, Run @, NoRaising true;

;;        ----------------------------------------------------------------------------
;;         OPERATING ON (EXTERNAL AND INTERNAL) OBJECTS
;;        ----------------------------------------------------------------------------

;; Requesting execution of an operation on an object
;; -----------------------------------------------------------

evtReq(ev, id) :: ObjMgrReq ev, CaaIdentity id
                  -> Run @, CaaIdentity id;

commit :: => -> ExecCommit @;

abort :: =>  -> ExecAbort @;

execCommit :: =>
        ExecCommit @, ExtObj o ->
        ExecCommit @, ObjMgrReq (evt o commit []);

execAbort :: =>
        ExecAbort @, ExtObj o ->
        ExecAbort @, ObjMgrReq (evt o abort []);

;;                ------------------------------------------------
;;                 ROLE PROGRAM INTERPRETOR
;;                ------------------------------------------------

;; Evaluation rules
;; -------------------

;; INSTRUCTION  "<instr1> ; <instr2>"

     eval :: =>
        Run @, Instr (Sequence i1 i2), InstrSnd i3 ->
        Run @, Instr i1, InstrSnd (Sequence i2 i3);

     eval ::
```

```
      !(i = Iend) =>
      Run @, Instr Iend, InstrSnd i ->
      Run @, Instr i, InstrSnd Iend;

   eval :: =>
      Run @, Instr Iend, InstrSnd Iend ->
      Run @, Outcome normal;

;; INSTRUCTION "Call <roleName>(<e1>,..,<en>) Of <caaName>"

   eval :: =>
      Run @, Instr (CallRole r withArgs args) ->
      RoleToCall r args;

;; INSTRUCTION " If <cond> Then <instr1> Else <instr2>"

   eval ::
      (subst cond c) = true =>
      Run @, Instr (If cond Then i1 Else i2), Ctxt c ->
      Instr i1, Ctxt c, Run @;

   eval ::
      (subst cond c) = false =>
      Run @, Instr (If cond Then i1 Else i2), Ctxt c ->
      Run @, Instr i2, Ctxt c;

;; INSTRUCTION "Raise <exceptionName>(<e1>,...,<en>)"

   eval :: =>
      Run @, Instr (Raise (ex! en exps)),
      Ctxt c, NoRaising false ->
      Suspended @, Instr Iend,
      Ctxt c, NoRaising false,
      RaisedExcept (ex! en (subst exps c));

   eval :: =>
      Run @, Instr (Raise e), NoRaising true ->
      Run @, Instr (Signal abort), NoRaising true;

;; INSTRUCTION "Signal <exceptionName>(<e1>,...,<en>)"

   eval :: =>
      Run @, Instr(Signal e), InstrSnd i,
      NoRaising b ->
      Instr Iend, InstrSnd Iend,
      NoRaising true, Outcome e;

;; INSTRUCTION "Assign <exp> to <x>"

   eval :: =>
      Run @, Instr (Assign exp To x), Ctxt c ->
      Run @, Instr Iend, Ctxt (assign (subst exp c) to x within c);

;; INSTRUCTION "Execute <o.m(e1,...,en)>"

   eval :: =>
      Run @, Instr (Execute (evt o m exps)),
```

```
    Obj o, Ctxt c ->
    ObjMgrReq (evt o m (subst exps c)),
    Obj o, Ctxt c;
```

Where

```
    args: objlist;
    c : environ;
    b, cond : bool;
    e: exception;
    en : exceptName;
    ev, ev1, ev2 : event;
    exp : expr;
    exps : exprlist;
    i, i1, i2, i3 : instruction;
    id : caaId;
    m : methName;
    ocaa, oext, ol, olist: objlist;
    o : objType;
    outc: outcome;
    r: roleType;
    x : varName;
```

End `Role;`

B.4 Specification of class IntExtObject

```
;;==================
Class IntExtObject;
;;==================

Interface

 Use ObjVersion;

 Type
   objType, varobjType;

 Subtype
   varobjType -> objType;

 Creation
   createObj _ : objVersionType;

Body

 Use ObjState, Name, Expression, CaaIdentity,
     Event, ListOfVersions, Role;

 Place
   VersionTree _ : objVersionType ;

 Transition
   serveRequest ;

 Axioms
    createObj v :: => -> VersionTree v;

    serveRequest With
        (r.evtReq(evt(Self,m,args), i)
         .. v.apply(evt(Self,m,args), i)) :: =>
        VersionTree v -> VersionTree v;

    Where
        r : roleType;
        v, vv : objVersionType;
        m: methName;
        args: exprlist;
        i: caaId;

End IntExtObject;


;;================
Class ObjVersion;
;;================

Interface

 Use CaaIdentity, ListOfVersions,
     Event, ObjState;
```

Type objVersionType;

Methods
apply _ _ : event caaId;

Creation
createObjVersion _ _ : caaId objVersionType;

Body

Use Bools;

Places
```
LocalVid _ : caaId;
ToExecute _ : event;
Versions _ : versionList;
LocalState _ : objState;
InitState _ : objState;
Parent _ : objVersionType;
Ended _ : bool;
ApplyAll _ _ : event versionList;
```

Methods
```
update _ : objState;
localState _ : objState;
startApplyAll _ : event;
```

Transitions
```
execute;
applyAll;
```

Axioms

;;
;; To be defined in the inheriting class according to the specific
;; behavior of the object for each of its methods.
;;
;;   createObjVersion i p :: =>
;;   -> LocalVid i, Versions [], InitState s,
;;   LocalState s, Parent p;

;;
;; Apply(e,i) identifies the version of the object state which must be modified.
;; It applies the event's effects to the state.

```
apply(ev, i)  ::
    i = ii
    => LocalVid ii, Versions vl ->
       LocalVid ii, Versions vl, ApplyAll ev vl;

apply(ev, i) With v.createObjVersion((prefx i ii), Self)
                .. v.apply(ev,i) ::
    (!(i=ii)) & (find(prefx(i,ii),vl) = vers(id*))
    => LocalVid ii, Versions vl ->
       LocalVid ii, Versions [vers(prefx(i,ii),v) |  vl] ;
```

```
    apply(ev, i) With v.apply(ev,i) ::
        (!(i=ii)) & (find(prefx(i,ii),vl) = vers(i,v))
        => LocalVid ii, Versions vl ->
           LocalVid ii, Versions vl;

;;
;; applyAll(ev) applies the event to the selected state
;; and to all its descendants.
;;

    startApplyAll(ev) :: Versions vl -> Versions vl, ApplyAll ev vl ;


    applyAll :: =>
        ApplyAll ev [] -> ToExecute ev ;
    applyAll With v.startApplyAll(ev) :: =>
        ApplyAll ev [vers(i,v) | vl] -> ApplyAll ev vl ;

;; To be defined in the inheriting class according to the specific
;; behavior of the object for each of its methods.
;;
;;   execute :: =>
;;      ToExecute (evt o m1 args), LocalState s -> LocalState ss;
;;   execute :: =>
;;      ToExecute (evt o m2 args), LocalState s -> LocalState ss;

    execute With p.localState(s) .. p.update(ss) ::
        eqList(s, sss) = true =>
        Parent p, LocalState ss, InitState sss,
        Ended false, ToExecute (evt o commit args) ->
        LocalState ss, InitState sss, Ended true ;

    execute :: =>
        ToExecute (evt o abort args), Ended false -> Ended true;

    execute :: =>
        ToExecute ev, Ended true -> Ended true;

    update(ss) :: => LocalState s -> LocalState ss;

    localState(s) :: => LocalState s -> LocalState s;

    Where
      ev : event;
      i, ii: caaId;
      v: objVersionType;
      vl : versionList;
      args : exprlist;
      o : objType;
      p : objVersionType;
      s, ss, sss : objState;

End ObjVersion;
```