COALA - A Formal Language for Coordinated Atomic Actions*

Julie Vachon, Didier Buchs, Mathieu Buffo
and Giovanna Di Marzo Serugendo
Software Engineering Laboratory (LGL-DI)
Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland.

Brian Randell, Sascha Romanovsky, Robert Stroud and Jie Xu
Department of Computing Science
University of Newcastle upon Tyne, Newcastle upon Tyne, UK.

Abstract

A Coordinated Atomic Action (CA action) is a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting objects in distributed object-oriented systems. They constitute a very interesting concept for the structured development of fault-tolerant distributed applications. To formalize the design of CA actions, this paper introduces a new language called COALA (COordinated Atomic actions LAnguage). COALA provides both a concrete syntax to write CA actions and a semantics which formally explains the concept. The semantics is given in the formal object-oriented specification language CO-OPN/2. COALA can thus benefit from the formal techniques developed around CO-OPN/2 and use them for the validation and the test of applications written with COALA CA actions.

*This work has been sponsored partially by the Esprit Long Term Research Project 20072 "Design for Validation" (DeVa) with the financial support of the OFES (Office Fédéral de l'éducation et de la Science), and by the Swiss National Science Foundation project "Formal Methods for Concurrency".

1 Introduction

General concepts

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 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

The aim of this paper is to describe a new formal specification language, COALA (CO-ordinated Atomic actions LAnguage), specifically developed to outline a precise semantics of the CA actions and to allow complex concurrent systems to be formally designed and specified using CA actions. This language includes a minimal set of instructions capturing the essence of CA actions; it has been developed as an extension of the CO-OPN/2 language. Its semantics has also been 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]).

This paper introduces COALA and demonstrates how it can be applied. The next section presents the specification language CO-OPN/2 which is used to express the COALA semantics. Section 3 describes in detail the main concepts of CA actions (roles, exception handling, objects, etc.) and the way in which they are included into COALA. Section 4 gives the complete description of the COALA syntax. The following section (Sect. 5) presents an overview of COALA semantics in terms of CO-OPN/2 objects. Finally, Section 6 presents a small banking example which illustrates how the language can be used for program design based on CA actions

2 Specifying Object Systems with CO-OPN/2

This section presents a short introduction to the concurrent O-O specification language called CO-OPN/2 (Concurrent O-O Petri Nets), which will be used to express the principles of COALA semantics. CO-OPN/2 is a formalism developed for the specification and design of large O-O concurrent systems ([Bib97], [BBG97a] and [BBG97b]). Such system consists of a possibly large number of entities, which communicate by triggering parameterized events (sending messages). The events to which an object can react are also called its methods. The behavior of the system is expressed with algebraic Petri nets. A CO-OPN/2 specification consists of a collection of two different kinds of modeling 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 internal state and provides the outside with various services. Cooperation between objects is performed by means of a synchronization mechanism, i.e. each event may request synchronization with the methods of one or of a group of partners using a synchronization expression. Three synchronization operators are defined: "//" for simultaneity, ".." for sequence, and "+" for alternative. The syntax of the behavioral axiom that includes synchronization is

$$\begin{array}{lll} \text{BehAxiom} & ::= & \textit{Event}\ [\textbf{With}\ \text{SynchroExpression}\ ]\ \textbf{::} \\ & & [\textit{Condition} =>]\ \textit{Precondition} -> \textit{Postcondition} \end{array}$$

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).

3 Fundamental concepts of CA actions

Roles

A CA action is viewed as a collection of roles, each of them being associated with a portion of code (a subprogram). The ultimate goal of a CA action consists in coordinating its roles so as to coherently manage all the system's software entities, i.e. the system's objects.

An execution thread enters a CA action by activating the role it wants to execute. When each role has been activated, the CA action starts and each thread thus executes its role. This execution is coordinated by the CA action which sees to it that all ACID properties, ensuring the consistent state of objects, are respected.

Objects

More precisely, the CA action concept defines two kinds of objects, namely *internal* and *external* objects. An internal object is an object local to CA action; it is shared between some or all its roles; it is used for the coordination of the roles or for the internal computation of a role. An external object is a global object which can be accessed by the roles of different CA actions (according to the visibility constraints). External objects may also be used for the coordination of the roles; operations on these objects are constrained by the ACID properties.

As explained later on, every internal object of an enclosing CA action may be used as an external object in a nested CA action. External objects may be shared simultaneously by several CA actions but the effect of the operations (read/write) applied to them must be the same as if the CA actions had been executed one after another. In other words, the schedule of operations applied to external objects must be *serializable*.

Outcomes

The execution of a CA action actually consists in coordinating the execution of the subprograms associated to its roles. A CA action always ends with one of the following outcomes:

- **Normal outcome**. Indicates that the ACID properties were satisfied during execution and the CA action commits its operations;
- **Exceptional outcome**. Indicates that the ACID properties were satisfied but an exception has been signalled to the enclosing CA action;
- **Abort outcome**. Indicates that the CA action has aborted and has undone its operations while satisfying the ACID properties;
- **Failure outcome**. Indicates that a major problem occured, preventing the CA action not only from commiting but also from aborting (i.e. the CA action ends without guarantying the satisfaction of the ACID properties).

All the roles of the CA action end with the *same* outcome; in the case of an *exceptional outcome* all the roles signal the *same* exception to the enclosing CA action.

Exceptions and Handlers

Two types of exceptions may be found in CA actions: *internal* and *interface* exceptions. Internal exceptions (noted $\{e_1, \ldots, e_n\}$) are totally local to a given CA action, which must therefore handle them on its own ; each role of a CA action has a set of subprograms (called *handlers*) to handle these exceptions or a combination of them. Internal exceptions can be *raised* by roles. When some roles of a CA action raise different internal exceptions concurrently, a resolution algorithm (using a resolution graph) determines which handler must be activated by the roles so as to handle the concurrent exceptions. Let's however remark that handlers are not authorised to *raise* internal exceptions.

As for interface exceptions (noted $\{\epsilon_1, \ldots, \epsilon_m\}$), they are *signalled*[1], that is to say they are propagated to the enclosing CA action. Both roles and handlers can signal interface exceptions. The signalled exceptions are propagated and thus correspond to *raised* exceptions at the level of the enclosing CA action.

Consider two CA actions `A` and `B`, such that `A` is nested in `B`. Roles of `B` can call roles in the nested CA action `A`. It is worth noting that interface exceptions of `A` must correspond to internal exceptions of `B`. For that reason, the set of internal exceptions $\{e_1, \ldots, e_n\}$ of `B` implicitly contains the set of interface exceptions $\{\epsilon_1, \ldots, \epsilon_m\}$ of `A`.

In addition, two generic interface exceptions are made available to all CA actions: the *Abort* exception and the *Fail* exception.

CA actions' behaviour

The execution of a CA action always corresponds to one of the following scenario:

- Each role executes and ends successfully. The CA action ends with a normal outcome;
- Some of the roles concurrently raise *internal* exceptions during the execution. These raised exceptions can be identical or different (recall that they can correspond to exceptions signalled by nested CA actions). In any case, the CA action uses a *resolution graph* to decide which exception handler has to be executed to cope with these concurrent exceptions. All the roles of the CA action are informed of the exceptions raised and of the exception handler which they must start executing. The following cases may occur:
  - If all the exception handlers end successfully, the CA action thus ends with a normal outcome;
  - If some of the exception handlers signal an interface exception to the enclosing CA action and if these exceptions are the *same*, then the underlying system forces all the handlers to signal this exception to the enclosing CA action. The CA action ends with an exceptional outcome;

[1] *Raising* an exception implies that some resolution and handling phases follow, while *signalling* an exception entails stopping the current execution and propagating the exception at a higher level.

  - If some of the exception handlers signal an interface exception to the enclosing CA action but these are *different*, the underlying system tries to perform an abort. If it succeeds the CA action ends with the abort outcome and all the roles signal an *Abort* exception to the enclosing CA action; if the abort operation fails, the CA action ends with the failure outcome and a *Fail* exception is signalled to the enclosing CA action.

    The underlying system behaves the same way if an exception is raised during the execution of a handler. This case may occur when a nested CA action, called during the execution of a handler, signals an exception which is consequently raised at the handler level. Since raised exceptions can't be solved in handlers, the underlying system must abort the action.

- Some of the roles signal an interface exception and these exceptions are all the same. In this case, the underlying system forces all the roles to signal this exception to the enclosing CA action. The CA action ends with an exceptional outcome;
- Some of the roles signal an interface exception but these exceptions are different. In this case, the underlying system performs an abort operation and if it succeeds all the roles signal an *Abort* exception; if it fails, they all signal a *Fail* exception;
- A role signals an interface exception while another role raises an internal exception. The raised exception is ignored, and the CA action continues executing accordingly to one of the two last cases explained above.

Pre and Post Conditions

A *pre-condition* and a set of *post-conditions* (i.e. each outcome, normal or exceptional, has an associated post-condition) can be inserted in a CA action interface as assertions about the behaviour of the CA action. These assertions entail a CA action to fulfill the following rules:

- A CA action can't start unless its pre-condition is satisfied;
- If the post-condition associated to the outcome of the exiting CA action is not satisfied, the CA action must abort.

The next section presents a description of COALA (COordinated atomic Action LAnguage), a formal specification language for CA actions. This language allows the description of CA actions, along with their roles, exceptions and handlers.

4 COALA Syntax

COALA is the formal language used for the design of CA actions. COALA's *instructions* are used to describe the coordinated behaviour of threads executing the CA action (description of roles and exception handlers). COALA also introduces *declarations* (1) giving the name of exceptions which may be raised or signalled, (2) defining the exception resolution graph, (3) stating the name of objects on which the threads may operate.

COALA relies on the specification language CO-OPN/2 to describe the behaviour of objects (internal and external) implied in CA actions.

4.1 Interface and Body

The COALA definition of a CA action is made of two parts: an *Interface* section, which is visible to other CA actions, and a *Body* section, which is private and hidden to the outside world.

The Interface of a CA action defines:

- the *name* of the CA action;
- the list of its (parameterised) *roles*;
- the list of CO-OPN/2 modules (adts and classes) which are being used in the Interface part;
- the list of the (parameterised) *interface exceptions* which the CA action can signal to an enclosing CA action.
- the *pre-condition* which the CA action should satisfied before it begins. At the moment, this pre-condition has no meaning in COALA's formal semantics. It simply plays the role of a structured comment and is thus not verified at execution time.
- for each possible outcome (normal or exceptional), the *post-condition* which the CA action should satisfied before exiting. As for pre-conditions, post-conditions are simply considered as structured comments at the moment.

The Body of a CA action defines:

- the list of CO-OPN/2 modules (adts and classes) which are being used in the Body part;

- the list of nested CA actions;
- the list of the CA action's internal objects. The behaviour of these objects are specified in separate CO-OPN/2 specification modules;
- the list of (parameterised) *internal exceptions* that roles can raise within a CA action. The interface exceptions of all nested CA actions are implicitly considered part of this list. It is not necessary to mention these exceptions here since it is easy to retrieve their name given the list of nested CA actions declared in the body;
- the list of the (parameterised) *exception handlers* of the CA action;
- the *resolution graph* of the CA action. In COALA, this graph lists the combinations of internal exceptions which can be raised concurrently, together with the handler which must be activated in each case. If a combination of internal exceptions can't be found in the list during execution, it is that no handler was foreseen to handle the case : an abort exception is therefore signalled by the underlying system;
- a *Where* field declaring local variables and their types;
- a *specification for each role*, which is comprised of:
  - the *parameterised name* of the role;
  - an *instruction block* (subprogram);
  - a *Where* field declaring local variables and their type;
  - a *specification for each exception handler*, containing its *parameterised name*, an *instruction block*, and a *Where* field.

4.2 Roles and Handlers

The behaviour of a role or a handler is described by an *instruction block*, that is a sequence (which may be empty) of instructions separated by commas. COALA's instructions are listed and explained hereafter. They may contain variables but also expressions, conditions and synchronizations referring to CO-OPN/2 specifications and terminology (c.f. [Bib97]).

4.2.1 COALA variables, expressions, conditions and synchronizations

- **Variable**. Name to which a (typed) value is associated. A variable must be in the scope of the *instruction block* where it is used. Variables are declared in the `Where` field of a role, a handler or a CA action's body.

According to the context where they are used, COALA variables may behave like both classical and logical variables. When used in COALA's instruction `Execute`, variables behave as logical variables and thus follow unification rules. Otherwise, they behave as classical variables and different values can be assigned to them.

- **Expression**. As mentioned, COALA uses CO-OPN/2 specifications for the definition of data types and object classes. The global signature of these CO-OPN/2 specifications is noted Σ. An expression a is thus simply a term written over this sorted signature Σ and over some sorted set $Vars$ of declared COALA variables (i.e. $a \in T_{\Sigma,Vars}$).

- **Condition**. This notion refers to CO-OPN/2 's definition of condition. Conditions are expressed with the usual equality and boolean predicates. Equality predicates apply to expressions ($a_1 = a_2$). Composed conditions ared formed using predicates ! for negation (! *condition*), & for conjunction (*condition*$_1$ & *condition*$_2$) and || for disjunction (*condition*$_1$ || *condition*$_2$).

- **Synchronization**. A synchronization[2] is a term noted $o.m(a_1, \ldots, a_n)$ where
  - o is the name of a CO-OPN/2 object declared in the `Object` field of a CA action or is a COALA variable refering to a CO-OPN/2 object.
  - m is a typed method defined in the specification of the CO-OPN/2 class to which o belongs.
  - $a_1, \ldots, a_n$ are expressions whose types are consistent with the type of object o's method m.

4.2.2 COALA's instructions

An *instruction block* (*instructionBlock*) is written:

`begin` *instruction* (; *instruction*)* `end`

An instruction (*instruction*) is either empty or is one of the followings:

- `Assign` a `To` v

  This instruction assigns *expression* a to *variable* v.

[2]Reference is here made to CO-OPN/2 synchronizations which thereupon do not differ from usual object method calls, apart from relying on unification for the evaluation of the arguments.

- `Execute` $o.m(a_1, \ldots, a_n)$

  This instruction tries to execute *synchronization* $o.m(a_1, \ldots, a_n)$. If method m of the object referenced by o is firable given parameters $a_1, \ldots, a_n$ and according to the corresponding CO-OPN/2 specification, then the instruction succeeds[3]. If m is not firable then an internal exception[4] is raised.

- `If` c `Then` $instructionBlock_1$ `Else` $instructionBlock_2$

  If condition c is true (according to the model of the given CO-OPN/2 specification), then $instructionBlock_1$ is executed. If c is false, then $instructionBlock_2$ is executed.

- `Raise` $exceptionName(a_1, \ldots, a_n)$

  This instruction allows a role to *raise* an internal exception within a CA action. $exceptionName$ must be the name of an internal exception defined in the body of the CA action. $a_1, \ldots, a_n$ are *expressions* which parametrise the exception raised. A resolution phase and an exception handling phase normally follow the raising of an exception.

- `Signal` $exceptionName(a_1, \ldots, a_n)$

  This instruction allows a role or an exception handler to *signal* an interface exception to an enclosing CA action. The name of the exception is either one of the exceptions defined in the interface of the CA action, or the *Abort* or *Fail* exception. When signalling an exception, a role/handler interrupts its execution and waits for the other roles/handlers to end so as to pass the acceptance test with the exceptional outcome signalled.

- `Call` $roleName(a_1, \ldots, a_n)$ `Of` $caaName$

  This instruction allows the activation of role $roleName$ in (nested) CA action $caaName$. If an instance of $caaName$ already exists and its role $roleName$ hasn't yet been fulfilled, then $roleName$ is activated with its actual parameters $a_1, \ldots, a_n$ passed on. Otherwise, the underlying system first creates a new instance of $caaName$, (with its own internal objects[5]), before activating the role $roleName$.

  The actual parameters $a_1, \ldots, a_n$ are expressions which may contain variables referring to objects. In this case, these objects are considered as "external" by the role which is called ($roleName$) and will operate on them.

  Note that the role/handler executing the `call` instruction must wait for the outcome of $caaName$ before pursuing its own execution.

[3]Note that unbound variables may take new values according to the unification principle which is applied in CO-OPN/2 .

[4]i.e. a predefined exception raised by the underlying system

[5]When an instance of CA action is destroyed, its internal objects are also destroyed

4.3 COALA's Concrete syntax and Statics Semantics

The concrete syntax of COALA is defined by a BNF grammar, which can be found in appendix B. To maintain a coherent and uniform set of languages, COALA's syntax has been designed so as to be similar to the concrete syntax of CO-OPN/2 .

Likewise, the statics semantics (i.e. the typing system) of COALA is consistent with the static semantics of CO-OPN/2. In fact, COALA uses the same typing system as CO-OPN/2; types used by COALA are CO-OPN/2 types, and COALA supports subtyping just like CO-OPN/2 does. Subtyping can be used anywhere in a COALA program where a typed expression appears. For instance, if a role in a CA action accepts a parameter of type t, then it can also accept any parameter of type t', as long as type t' is a subtype of type t according to the CO-OPN/2 specification used by the CA action.

A tool, named *coalacheck*, has been developed to check the correctness of COALA source texts. This tool verifies the COALA syntax, as well as its static semantics. The banking example in the next section has thus been verified with it. *Coalacheck* is based on the checking tool developed for CO-OPN/2 source texts.

5 COALA's Semantics

5.1 An Overview

The following lines should give the reader a short overview of COALA's semantics. Details can be found in [VB98]. The object-oriented specification language CO-OPN/2 , which is based on Petri nets and algebraic data types, was chosen as the target language for COALA's semantics. The aim of this work thus consisted in providing translation schemes of COALA constructions to CO-OPN/2 objects. There are no formal translation rules. The translation from COALA to CO-OPN/2 rather consists in applying translation schemes and specializing the basic CO-OPN/2 templates proposed so as to describe the particular behaviour of each COALA program.

Efforts were spent writing CO-OPN/2 classes to be used as templates for the semantic description of CA actions and their roles. Hence, two main abstract classes, named `Caa` and `Role`, have been defined to specify the fundamental bahavioral properties common to, respectively, any CA action and any role. The semantics of particular CA actions and roles can thus be expressed by creating new sub-classes which inherit from the basic abstract classes (`Caa` or `Role`) and which define new axioms to specify these specific behaviors.

This inheritance relationship is illustrated on Figure 1. This figure also shows the subtyping relation which exists beween the CO-OPN/2 objects representing CA actions or roles. Among others, subtyping allows to gather under a common subtype (e.g. `Role_of_Caa1`, `Role_of_Caa2`, etc.) all the `role` objects of a given CA action. Since CA actions are entities which coordinate roles, a clientship relation (Fig. 2) exists between classes `Caa` and `Role`. Indeed, as it will be explained in the semantics, `caa` objects use services (methods) provided by `role` objects. Methods of `role` instances can be seen as hands held to `caa` objects allowing for their coordination.

Besides classes `Caa` and `Role`, many other CO-OPN/2 classes and abstract data types were defined to explain the interpretation of role programs (COALA instructions), the evaluation of CO-OPN/2 expressions used in COALA, the handling of exceptions, the consistent access to external objects, etc. Figure 2 also illustrates part of the "Use" relation which exists between CO-OPN/2 modules.

Considering the mechanisms and the properties specific to CA actions, the next subsections explain with more details the semantics of COALA CA actions and roles, as well as their translation into object instances of class `Caa` and `Role`.

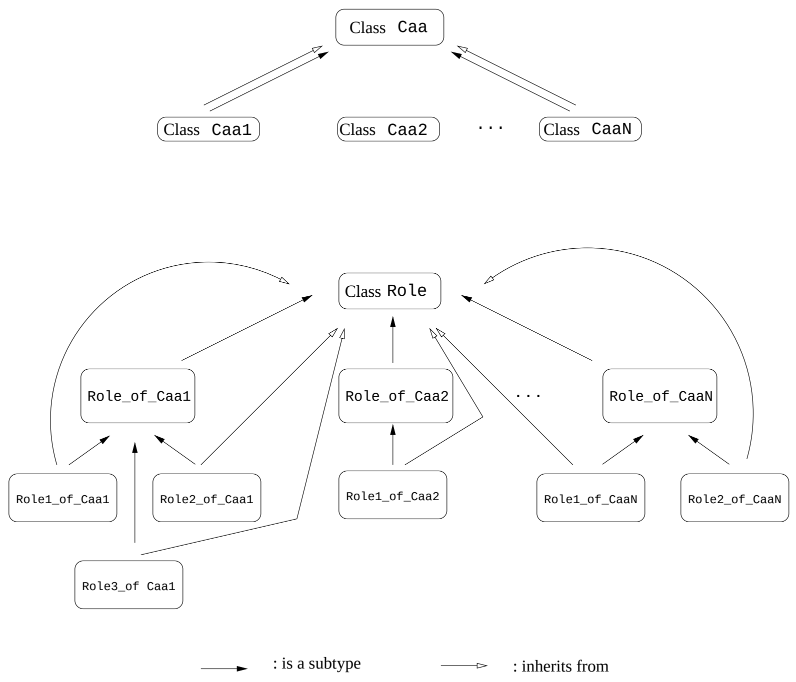

Figure 1: Inheritance and subtyping hierarchies

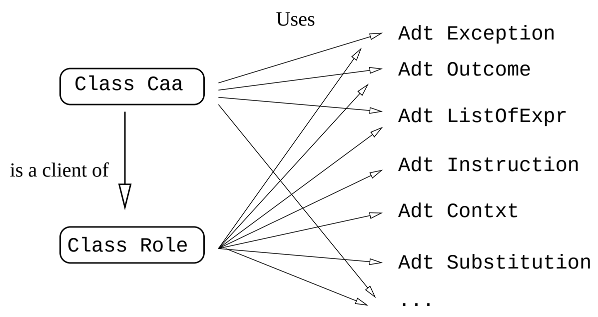

Figure 2: *Clientship* and *use* hierarchies

5.2 Translating COALA's CA actions to CO-OPN/2 objects

COALA's CA actions are coordinating entities which are represented in the CO-OPN/2 semantics by object instances of an abstract class named `Caa` (cf. Appendix C.1). This abstract class specifies the general behavior of all COALA CA actions. Specific CA actions with their particular behavior are described by subclasses which inherits from class `Caa`.

The principal coordinating activity of a CA action is ensured by the internal transition `syncRoles` :

```
syncRoles With
    Self.startRoles(rlist, clist, caaObj) ..
    Self.acceptTest(rlist, outc) ..
    Self.endRoles(rlist, clist, outc) ::
  Objs caaObj, Roles rlist -> Objs caaObj, Roles rlist;
```

According to the semantics of CO-OPN/2 , transition `syncRoles` can fire if and only if the *synchonization expression*, which follows the keyword `With`, can itself fire. Let us recall that operator "`//`" specifies a simultaneous execution of two events, while operator "`..`" denotes a sequential execution. The reader can refer to [Bib97], [BBG97a] and [BBG97b] for details about CO-OPN/2 's syntax and semantics.

Transition `syncRoles` thus consists in (1) synchronizing the beginning of all the roles of the CA action (`startRoles`), then (2) coordinating the acceptance test (`acceptTest`) and finally (3) signalling the outcome of the ending roles (`endRoles`).

Parameters `rlist` and `caaObj` can be considered as *input* values which respectively denote the list of internal objects and the list of roles of the CA action. Parameters `outc` and `clist` are *output* values denoting respectively the CA action's outcome and the list of threads (roles) which have called and synchronized with a role in the CA action. (N.b. `rlist` and `clist` thus have the same length.)

Here follows some details about each of the three coordination activities mentioned above and which are specified by a synchronization sub-expression in the `With` part of the `syncRoles` axiom:

1. The recursive method `startRoles` specifies that a CA action must start all the roles simultaneously. For this, the CA action must synchronize sequentially with first an object representing a calling thread (`c.callRole`) and then the object reprensenting the role to be started (`r.start`). This is done simultaneously for all the roles of the CA action i.e. those which are listed in argument (`r'rlist`). If there are many simultaneous calling threads for a same role, one of them is chosen randomly and the

variable "c" takes the identity of that object by unification.

```
startRoles([], [], caaObj):: ->;
startRoles((r ' rlist), (c ' clist), caaObj)
 With (c.callRole(r, arg) .. r.start(arg, caaObj))
      // Self.startRoles(rlist, clist, caaObj):: ->;
```

2. Method `acceptTest` specifies how a CA action coordinates the acceptance test. All the roles must agree on the final outcome. Thus, according to the second axiom of method `acceptTest`, if all the roles end with the same outcome (be it normal or exceptional), the acceptance test returns the value of this outcome. The third axiom specifies that if roles have different outcomes but none of them failed, then the acceptance test returns `abort`. The last axiom considers the case where at least one role fails and thus entails the acceptance test to return the outcome `fail`.

```
acceptTest((r ' []), outc) With r.outcome(outc):: ->;
acceptTest((r ' rlist), outc1)
 With (r.outcome(outc1) // Self.acceptTest(rlist, outc2)) ::
   eq(outc1, outc2) and not(empty?(rlist)) = true => ->;
acceptTest((r ' rlist), abort)
 With (r.outcome(outc1) // Self.acceptTest(rlist, outc2)) ::
  not(eq(outc1, outc2)) and not(eq(outc1, fail))
   and not(eq(outc2,fail)) and not(empty?(rlist)) = true => ->;
acceptTest((r ' rlist), fail)
 With (r.outcome(outc1) // Self.acceptTest(rlist, outc2)) ::
  (eq(outc1, fail) or eq(outc2, fail))
   and not(empty?(rlist)) = true =>  ->;
```

3. Method `endRoles` is responsible for transmitting the roles' outcomes (normal or exceptional) to each of the calling threads simultaneously. As specified by the third axiom of `endRoles`, if the outcome is `abort`, each role must abort before it is signalled.

```
endRoles([], [], outc):: ->;
endRoles((r ' rlist), (c ' clist), abort)
 With (r.abort .. c.signalOutcome(abort))
      // Self.endRoles(rlist, clist, outc):: ->;
endRoles((r ' rlist), (c ' clist), outc)
 With c.signalOutcome(outc) // Self.endRoles(rlist, clist, outc) ::
   eq(outc, abort) = false => ->;
```

It is important to recall that transition `syncRoles` is performed only if all the sychronization sub-expressions sequentially succeed. This all-or-nothing semantics therefore guarantees that a CA action behaves as specified or does not execute at all.

Moreover, CA actions are also responsible for coordinating the resolution of concurrently raised exceptions. The axiom defining transition `solveExceptions` allows to specify this mechanism which consists in (1) collecting simultaneously all the exceptions concurrently raised by roles, (2) determining the *global* exception which corresponds to the set of raised exceptions according to the CA action's resolution graph, and (3) for each role simultaneously, activating the handler which must cope with this *global* exception.

These tasks are described by the methods appearing in the *synchronization expression* of transition `solveException`:

```
solveExceptions With
    Self.getExceptions(rlist, exlist) ..
    Self.solve(exlist, e) ..
    Self.handleExcept(rlist, e) ::
      eq(e, nil) = false => Roles rlist -> Roles rlist;
```

1. The recursive method `getExceptions` simultaneously asks every role the exception it has just raised (`r.exRaised`). The exception returned by a role is "nil" if it has raised no exception.

   ```
   getExceptions([], []):: ->;
   getExceptions((r 'rs), (e ' es))
        With r.exRaised(e)  // Self.getExceptions(rs, es):: ->;
   ```

2. Method `solve` proceeds to the exception resolution phase and thus determines the "global" exception which corresponds to the list of collected exceptions (cf. 1.). Since this method encodes the CA action's resolution graph, its axioms are thus different for each particular CA action. Only two axioms of method `solve` are common to all CA actions. (The others are defined in the subclasses.)

   ```
   solve(l, fail) :: fail isIn? l =>  ->;
   solve(l, abort) :: not (fail isIn? l) and (abort isIn? l) => ->;
   ```

   These axioms correspond to the abort and failure cases: if at least one role fails then the `fail` exception must be globally raised; if at least one role aborts but no role fails, then the `abort` exception is globally raised.

3. Method `handleExcept` tells each role simultaneously the "global" exception to be handled. All the roles thus activate their appropriate handler (`r.handle`).

   ```
   handleExcept([], _):: ->;
   handleExcept((r 'rs), e)
        With r.handle(e) // Self.handleExcept(rs, e) :: ->;
   ```

5.3 Translating COALA's roles to CO-OPN/2 objects

Roles are translated into CO-OPN/2 objects belonging to the abstract class `Role` which specifies the generic mechanisms and the basic behavior that any role must have. Part of the specification of class `Role` is given in Appendix C.2.

Specific roles are described by individual classes which inherit from the abstract class `Role`. These subclasses are specialized following some given scheme and according to the specific programs which the roles have to execute. Besides inheritance, other object-oriented features of CO-OPN/2 have been used in the definition of COALA's semantics. In particular, subtyping has been especially useful for gathering all the roles of a same CA action under a particular type (Fig. 1). Thanks to this last feature, a given `caa` object can refer to types to identify the `role` objects it coordinates.

As for `caa` objects, `role` objects also have methods and internal transitions specifying their behavior. The interface of class `Role` is the following:

```
Class Role ;
Interface
  Use Name, ListOfExpr, ListOfObjects, Outcome;
  Type roleType;
  Methods
     callRole _ _ : roleType objlist;
     outcome _ : exception;
     exRaised _ : exception;
     handle _ : exception;
     evtReq _ : event;
     objMgrAns _ : outcome;
  Create
    start _ _ : objlist objlist;
End Role;
```

For the most part, the methods declared in this interface are services used by `caa` objects to coordinate `role` objects and to direct their execution according to their state and evolution.

`Role` objects also have many hidden[6] methods and transitions. Their main internal transition is called `eval` and is used for interpreting COALA instructions. Figure 3 illustrates the evaluation of the instruction "`Call` $roleName(a_1, \ldots, a_n)$ `Of` $caaName$". The axiom describing the behavior of this instruction is the following:

[6]declared in the body section and not in the interface.

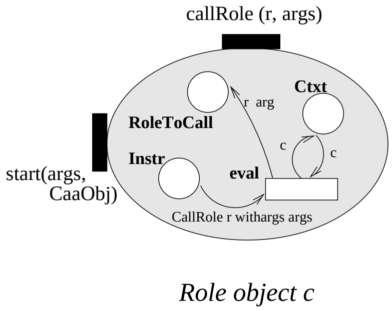

Figure 3: Role object c evaluating a `Call role` instruction

```
eval :: Instr (CallRole r withArgs args), Ctxt c,
         -> RoleToCall r (subst args c), Ctxt c;
```

As illustrated, the instruction is read from place `Instr`. To interpret it, the evaluation context (containing the values assigned to the variables) is required and is thus taken from place `Ctxt` and put back into it right after. When evaluating a `Call` instruction, transition `eval` puts the identity of the role to be called along with its evaluated parameters in place `RoleToCall`. These values will be fetch from this place as soon as the CA action object whose role is being called can synchronize with the method `callRole` of the calling object. This synchronization is illustrated by Figure 4.

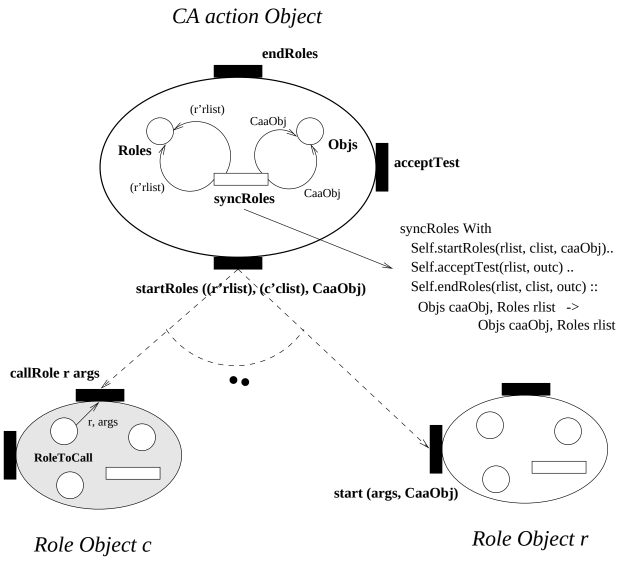

Figure 4: Synchronization between the *calling* role object c and the *called* role object r

In this figure, the `Call` instruction is performed by `Role` object "c" while the `role` object being called is identified by "r". As specified by the `eval` axiom above, the interpretation of this instruction consists in evaluating the arguments in the context and then putting the result together with the identity of the `role` object to be called (i.e. r) in place `RoleToCall`. Appropriate tokens in this place allow method `callRole` of `role` object c to eventually fire. It does indeed as soon as the method `startRoles` of the `caa` object coordinating role r, tries to synchronize sequentially with (1) the method `callRole` of object c and (2) the method `start` of object r.

`Role` objects have many other methods which are used to define the exception handling mechanism, the way roles operate on CA actions external objects, etc. These were not introduced here for the explanation of their axioms would go beyond the scope of this paper.

5.4 Other concepts of the semantics

Some important concepts of CA actions, such as the serializability of operations applied to external objects, haven't been explained here. A technical report ([VB98]) describing the complete semantics of COALA is available. Detailed explanations and all the corresponding translation schemes of COALA in CO-OPN/2 can be found in this report.

6 The banking example

This section introduces a small case study to illustrate how COALA's CA action concepts and syntax are used for the design of fault-tolerant distributed applications. The complete COALA specification of this example is given in Appendix A.

The banking example illustrates how a *joint withdraw*, implicating two clients and two joint accounts, can be specified using CA actions designed in COALA.

More precisely, the bank of this case study offers its clients a special kind of account called "joint account". A joint account is owned by two clients, called co-owners. Each owner is given a personal identification number (or pin) which he must use to identify himself.

The conditions applied to joint accounts and pins are the following.

Joint accounts

- A Withdraw operation on a joint account requires the authorisation of both co-owners.
- Money may be deposited on the joint account without any authorisation.
- The balance of the account may be consulted by both co-owners.

Personal identification numbers

- A client agrees for the execution of a transaction by giving his pin, which must henceforth be validated.
- A client is authorized to perform an operation if he can identify himself with a valid pin. If he makes a mistake, the operation is immediately aborted.

The COALA program corresponding to this banking example is made of two parts:

1. a CO-OPN/2 specification of the objects to be used (`IntegerContainer`, `PIN` and `Account`);
2. a COALA description of the required CA actions (`JointWithdraw`, `Withdraw` and `Wait`)

The COALA description itself consists of two principal CA actions (`JointWithdraw` and `Withdraw`) and of three complementary one (`WaitPIN`, `WaitInfoAmount` and `WaitReadAmount`) which are simply used to force the synchronization of threads. The following subsections give more details about the behavior of these particular CA actions.

CA action `JointWithdraw`

The `JointWithdraw` action presents two clients, named `client1` and `client2`. They are the co-owners of two joint accounts, `account_1` and `account_2`.

Each joint account has an appointed co-owner who takes care of the main transactions operated on the account: hence, `client1` is responsible for transactions on `account_1` while `client2` is responsible for transactions on `account_2`.

In CA action `JointWithdraw`, role `client1` describes the behavior of a client who wants to withdraw a certain amount of money out of one of two given joint accounts. More precisely, the money must be withdrawn from the account having the greatest balance. `client1` thus informs the other co-owner, `client2`, about the amount to withdraw. Each client consults his appointed account and tells the other how much money there is left. Each client henceforth knows on which account the money must be taken from. The client responsible for the account having the highest balance thus performs the withdraw by calling role `withdrawer` of CA action `Withdraw`. On his side, the other client calls role `partner` of this same CA action.

Moreover, if there is not enough money on a single account, the missing amount is drawn out the other account. If some money is still missing, the exception `NotEnoughMoney` must be signalled. Figure 5 describes the COALA interface of CA action `JointWithdraw`. Its body and the complete COALA program is given in the appendix. (Pre and post-conditions were not really necessary to solve the banking problem and thus have not been introduced in this example.)

As for Figure 6, it illustrates one of the normal (i.e. with no exception) execution of CA action `JointWithdraw`. Boxes delimit the execution part under the control of each CA action. Circles represent objects. As for "X" symbols, they denote `Execute` intructions in the role programs. Details about the CA actions which are nested in `JointWithdraw` are given in the next subsections.

```
Caa JointWithdraw;

Interface

    Use
        Account, Integers;      ;; This CAA uses the COOPN/2 modules
                                ;; specifying the object class representing
                                ;; accounts and the data type for integers.

    Roles
        client1 : accountType, integer;
                                ;; The first client is the one who
                                ;; determines the amount of money to
                                ;; be withdrawn. This role is thus
                                ;; parameterised by the account used by
                                ;; the client and the money he plans
                                ;; to withdraw.

        client2 : accountType;
                                ;; The second client works in
                                ;; partnership with the first client.
                                ;; This role is simply parameterised
                                ;; by the account the client uses.

    Exceptions
        NotEnoughMoney;         ;; This exception is signalled when
                                ;; there is not enough money on
                                ;; both accounts.

End JointWithdraw;
```

Figure 5: COALA interface of CA action `JointWithdraw`

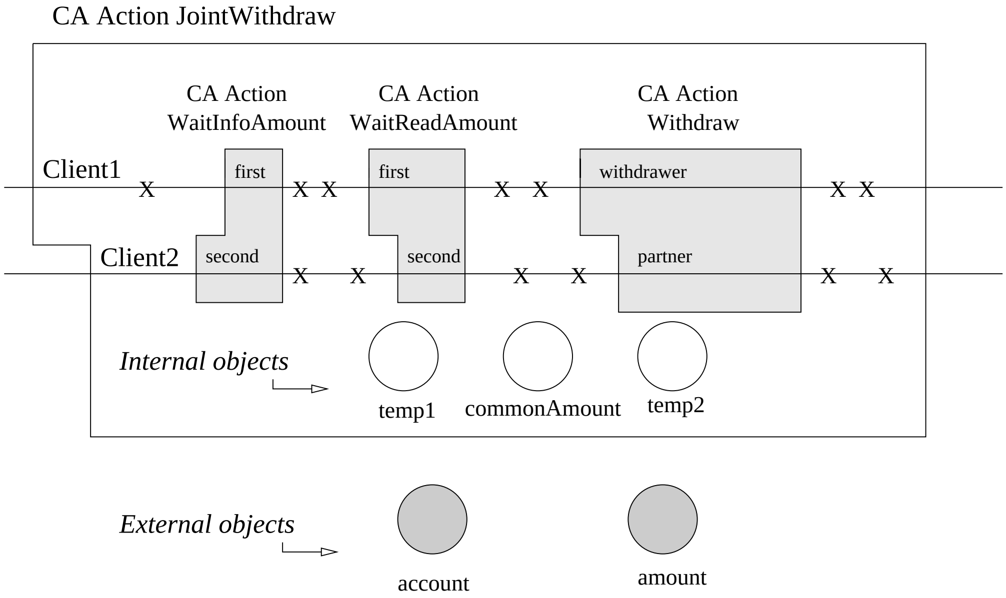

Figure 6: A (normal) execution of CA action `JointWithdraw`

CA action `Withdraw`

As mentioned, a withdraw operation requires the authorization of both co-owners. Hence, the client withdrawing the money (the `withdrawer`) must not only give is own pin but must also get the pin of the other co-owner (his `partner`).

Note that clients have a single try to enter their pin when they are prompted for it. If the wrong pin is entered, then an `Abort` exception is raised, and the whole CA action `Withdraw` is aborted.

If the pin validation process succeeds, two cases may occur:

1. There is enough money on the account. The required money is drawn out the account and the balance is updated.
2. There is not enough money on the account. All the money on the account is thus being drawn out (the balance is put to 0) and the exception `missing` is raised. The `withdrawer`'s exception handler takes up the execution and indicates the amount of missing money to the enclosing CA actions by assigning this amount to the external object `commonAmount`. If no problem occurs during this handling phase, CA action `Withdraw` simply ends with a normal outcome.

Figure 7 illustrates the normal execution of CA action `Withdraw`.

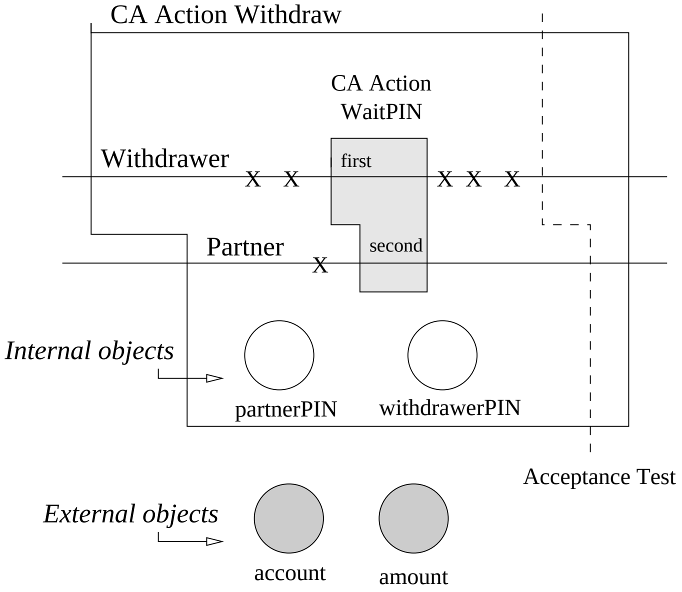

Figure 7: A (normal) execution of CA action `Withdraw`

CA actions `WaitPin`, `WaitInfoAmount` and `WaitReadAmount`

These CA actions are used to coordinate the work of two threads, more precisely to synchronize two threads. They all have the same shape: a first thread enters the CA action by calling one of the two roles; this thread is suspended until the other role is called by a second thread. Since the body of the roles are empty, these CA actions end (with a normal outcome) right after the initial sychronization of the roles.

7 Conclusions

The CA action concept constitutes an attractive and promising approach to structuring complex concurrent object-oriented systems and to providing their fault tolerance in a disciplined and rigorous way. It occupies a crucial place in the conceptual framework proposed by the DeVa project which concerns with the development of reliable applications and with applying advanced structuring techniques for improving our ability to validate the developed systems and to reason about their behaviour.

COALA is introduced as a formal language for designing and specifying systems which are developed using CA actions. It is based on object-oriented specification language CO-OPN/2 . A COALA formal semantics is described in terms of CO-OPN/2 objects. Writing the semantics of COALA, we had the chance to benefit from many of the highlevel features of CO-OPN/2 : synchronization expressions, object-oriented mechanisms, abstractions, etc. Among others, inheritance and subtyping helped to provide a structural and modular description of CA actions and their roles.

Since COALA's semantics is expressed in CO-OPN/2 , all existing development methods and tools built for CO-OPN/2 can be employed while writing COALA specifications ([BB97], [PBB98] and [SG98]). In addition to this a first COALA-oriented tool has recently been developed and is now integrated into the CO-OPN/2 set of tools: it allows checking both syntax and static semantics (i.e. the typing system) of the COALA specifications.

Further works include the development and the formalisation of the translation rules which compute the semantics of the COALA programs in terms of CO-OPN/2 object systems. A JAVA implementation of the CO-OPN/2 language should then allow for the simulation and the testing of COALA programs.

8 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.

[BBG97a] 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.

[BBG97b] O. Biberstein, D. Buchs, and N. Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In G. Agha and F. De Cindio, editors, *Advances in Petri Nets on Object-Orientation*, Lecture Notes in Computer Science. Springer-Verlag, 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).

[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.

[SG98] G. Di Marzo Serugendo and N. Guelfi. Using object-oriented algebraic nets for the reverse engineering of java programs: A case study. In *Proceedings of the International Conference on Application of Concurrency to System Design (CSD'98)*. IEEE Computer Society Press, 1998.

[VB98] J. Vachon and D. Buchs. The semantics of COALA in CO-OPN/2. Technical Report EPFL-DI No 98/300, Ecole Polytechnique Fédérale de Lausanne, CH-1015, Suisse, 1998.

[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 COALA specification of the banking example

A.1 Specification of Objects in CO-OPN/2

```
Abstract Class PIN;
;; Represents a Personal Identification Number.
;; This class is left abstract; its body is not described.
Interface
    Use Booleans;
    Type PINType;                  ;; The name of the type
    Method
        ;; Test for equality of PINs; the last parameter returns the
        ;; corresponding boolean value
        equals _ _ : PINType, boolean;

        ;; this methods prompts a user for a PIN code. The PIN object
        ;; is hence modified accordingly.
        queryForPIN;
;; Body not described.
End PIN;

Abstract Class Account;
;; Represents a joint account
Interface
    Use PIN, Integers, Booleans;
    Type accountType;              ;; The name of the type
    Methods
        ;; Test for the PINs validity. The last argument
        ;; returns true if the two PINS are valid for this account;
        ;; the boolean false is returned otherwise.
        correctPINs _ _ _ : PINType, PINType, boolean;

        ;; Return the balance
        balance _ : integer;

        ;; Balance increment and decrement
        increment _, decrement _ : integer;
```

Creation

;; The creation of a new joint account requires both PINs

```
newAccount _ _ : PINType, PINType;
```

Body

Places

```
PINs _ : PINType;
sum _ : integer;
```

Axioms

```
correctPINs (p1,p2, (b1 and b2) or (b3 and b4)) With
    (pin1.equals(p1,b1) // pin2.equals(p2,b2)) //
    (pin1.equals(p2,b3) // pin2.equals(p1,b4)) ::
  PINs p1, PINs p2 : -> PINs p1, PINs p2;

balance x :: sum x  -> sum x;

increment x :: sum y -> sum (y+x);

((y-x)>=0)=true => decrement x :: sum y -> sum (y-x);

newAccount p1 p2 :: -> PINs p1, PINs p2, sum 0;
```

Where

```
pin1, pin2, p1, p2 : PINType;
b1, b2, b3, b4 : boolean;
x,y : integer;
```

End `Account;`

Class `IntegerContainer;`

;; Container for an integer;

Interface

Use `BlackTokens, Integers;`

Type `integerContainer;`

Methods

;; Puts a new integer value in the container;

```
put _ : integer;
```

;; Reads the integer value in the container;

```
get _ : integer;
```

Body

Places

```
empty _ : blackToken;
```

```
    hold _ : integer;
Initial
    empty @;
Axioms
    put x :: empty @ -> hold x;
    put x :: hold y -> hold x;
    get x :: hold x : -> ;
Where
    x,y : integer;
End IntegerContainer;
```

A.2 Description of CA actions in COALA

```
;; This CAA allows two clients (co-owners of two joint accounts) to
;; agree on a banking operation, namely a withdraw operation on their
;; joint accounts; after this agreement step, both clients call a
;; nested CAA performing the withdraw itself. the clients decide to
;; withdraw money from the account having the greatest balance. If
;; there is not enough money on a single account, they try to get the
;; missing amount from the other account. To do this, both clients
;; enter another nested CAA which performs the withdraw on the other
;; account.

Caa JointWithdraw;
Interface
  Use
    Account, Integers;      ;; This CAA uses the COOPN/2 modules
                            ;; specifying the object class representing
                            ;; accounts and the data type for integers.
  Roles
    client1 : accountType, integer;
                            ;; The first client is the one who
                            ;; determines the amount of money to
                            ;; be withdrawn. This role is thus
                            ;; parameterised by the account used by
                            ;; the client and the money he plans
                            ;; to withdraw.
    client2 : accountType;
                            ;; The second client works in
                            ;; partnership with the first client.
```

```
                            ;; This role is simply parameterised
                            ;; by the account the client uses.
  Exceptions
    NotEnoughMoney;         ;; This exception is signalled when
                            ;; there is not enough money on
                            ;; both accounts.
Body
  Use Booleans;

  Use Caa Withdraw, WaitInfoAmount, WaitReadAmount;

  Object
    temp1, temp2, commonAmount: integerContainer;   ;; Define local objects

  Handler
    FailHandler;            ;; Default Handler
    Aborthandler;

  Resolution
    Abort                    -> Aborthandler;

  Role client1 (account, amount);

    Begin
      Execute commonAmount.put(amount);   ;; store the amount to withdraw

      Call first Of WaitInfoAmount;       ;; wait for / wake up client2

      Execute account.balance(money);     ;; Get the balance of
                                          ;; client1's account
      Execute temp1.put(money);
      Call first Of WaitReadAmount;       ;; wait for client2 to fill temp2
      Execute temp1.get(t1);              ;; fetch the values
      Execute temp2.get(t2);
      If ((t1>=t2)=true) Then Begin ;; client1 is the withdrawer
        Call withdrawer(account,commonAmount) Of Withdraw;
        Execute commonAmount.get(ca);
        If (ca > 0 = true) Then           ;; still some money to withdraw
                Call partner Of Withdraw;
      End
      Else Begin                          ;; client1 is the partner
        Call partner Of Withdraw;
        Execute commonAmount.get(ca);
        If (ca > 0 = true) Then           ;; still some money to withdraw
                Call withdrawer(account,commonAmount) Of Withdraw;
      End;
      Execute commonAmount.get(ca);
```

```
        If (ca > 0 = true) Then
            Signal NotEnoughMoney;             ;; still money missing
    End

    Where
        t1, t2, ca, money: integer;
        amount : integer;
        account: accountType;

    Handler FailHandler;
        Begin
            Signal Fail;
        End
    End FailHandler;

    Handler Aborthandler;
        Begin
            Signal Abort;
        End
    End Aborthandler;

End client1;


Role client2 (account);

    Begin
        Call second Of WaitInfoAmount;        ;; wait for amount to withdraw

        Execute account.balance(money);       ;; get the balance of
                                              ;; client2's account
        Execute temp2.put(money);

        Call second Of WaitReadAmount;        ;; wait for / wake up client1

        Execute temp1.get(t1);                ;; fetch the values
        Execute temp2.get(t2);
        If ((t2 > t1)=true) Then Begin        ;; client2 is the withdrawer
            Call withdrawer(account,commonAmount) Of Withdraw;
            Execute commonAmount.get(ca);
            If (ca > 0 = true) Then           ;; still some money to withdraw
                    Call partner Of Withdraw;
        End
        Else Begin                            ;; client2 is the partner
            Call partner Of Withdraw;
            Execute commonAmount.get(ca);
            If (ca > 0 = true) Then           ;; still some money to withdraw
                    Call withdrawer(account,commonAmount) Of Withdraw;
        End;
        Execute commonAmount.get(ca);
```

```
        If (ca > 0 = true) Then
            Signal NotEnoughMoney;              ;; still money missing
    End

    Where
        t1, t2, ca, money : integer;
        account : accountType;

    Handler FailHandler;
        Begin
            Signal Fail;
        End
    End FailHandler;

    Handler Aborthandler;
        Begin
            Signal Abort;
        End
    End Aborthandler;

  End client2;

End JointWithdraw;


;; The aim of this CAA is to allow two clients to withdraw a predefined
;; amount of money from a predefined account; it is supposed that both
;; clients already agree on the transaction before this CAA is called.

Caa Withdraw;

Interface

  Use
    Account, IntegerContainer; ;; This CAA uses the COOPN/2 modules
                               ;; specifying object classes representing
                               ;; accounts and integer containers.

  Roles
    withdrawer : accountType, integerContainer;
                             ;; The withdrawer is the role responsible
                             ;; for the withdraw. It is parameterised
                             ;; by the account from which to money has
                             ;; to be taken from and by an object
                             ;; containing the amount to withdraw.

    partner;                 ;; The partner has no parameters;
                             ;; The partner has no other mission then
                             ;; simply giving his authorisation for
                             ;; the withdraw by providing his PIN.
```

```
Body
    Use
        PIN, Booleans;              ;; This CAA uses the COOPN/2 modules
                                    ;; specifying the object class representing
                                    ;; PIN and the data type for booleans.
                                      ;; accounts and integer containers.
    Use Caa
        WaitPIN;                    ;; CAA WaitPIN is used to
                                    ;; synchronize the threads executing
                                    ;; the roles.
    Object
        withdrawerPIN: PINType;     ;; Definition of a local object
        partnerPIN: PINType;        ;; Definition of a local object
    Exceptions
        missing : integer, integerContainer;    ;; Not enough money on the
                                    ;; withdrawer account; parameters are
                                    ;; used to indicate the missing amount
                                    ;; and the object containing the amount
                                    ;; which should have been withdrawn by
                                    ;; this CAA.
    Handlers
        FailHandler;                ;; Default Handler
        missingHandler : integer, integerContainer;
    Resolution
        missing(missingAmount, am)      -> missingHandler(missingAmount,am);
    Where
        missingAmount : integer;
        am : integerContainer;
    Role withdrawer (account, amount);
        Begin
            Execute account.balance (balance);   ;; get the balance of the
                                                 ;; withdrawer's account
            Execute withdrawerPIN.queryForPIN;   ;; prompt the withdrawer
                                                 ;; for his PIN
            Call first Of WaitPIN;               ;; wait for the partner's PIN
            Execute account.correctPINs(withdrawerPIN, partnerPIN,b);
            Execute amount.get(a);               ;; get the amount to withdraw
            If (b=true) Then Begin
                If ((balance>=a) = true) Then Begin
```

```
            Execute account.decrement(a)
        End
        Else Begin
            Execute account.decrement(balance);
            Raise missing(a-balance,amount)
        End
    End
    Else Begin
        Signal Abort;                     ;; PIN error
    End
End

Where
    balance,a : integer;
    amount : integerContainer;
    account :        accountType;
    b:               boolean;

Handler FailHandler;
    Begin
        Signal Fail;
    End
End FailHandler;

Handler missingHandler(mAmount, amount);
    Begin
        Execute amount.put(mAmount);     ;; store the missing amount
    End
    Where
        mAmount: integer;
        amount : integerContainer;
End missingHandler;

End withdrawer;


Role partner;

    Begin
        Execute partnerPIN.queryForPIN;      ;; asks the partner's PIN
        Call second Of WaitPIN;             ;; awake withdrawer
    End

    Handler FailHandler;
        Begin
            Signal Fail
        End
    End FailHandler;

    Handler missingHandler (mAmount, amount);
```

```
            Begin
                                                    ;; does nothing
            End
            Where
                mAmount: integer;
                amount : integerContainer;
        End missingHandler;

    End partner;

End Withdraw;
```

CA actions used for Synchronization

```
Caa WaitPIN;

Interface

    Roles
        first;
        second;
Body
    Role first;
        Begin
        End
    End first;
    Role second;
        Begin
        End
    End second;
End WaitPIN;



Caa WaitInfoAmount;

Interface

    Roles
        first;
        second;
Body
    Role first;
        Begin
        End
    End first;
    Role second;
        Begin
```

```
        End
    End second;
End WaitInfoAmount;


Caa WaitReadAmount;

Interface

    Roles
        first;
        second;
Body
    Role first;
        Begin
        End
    End first;
    Role second;
        Begin
        End
    End second;
End WaitReadAmount;
```

B 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)*
```

C COALA's semantics of CA Actions and roles in CO-OPN/2

The abstract classes which follows are only partially described. See [VB98] for the complete CO-OPN/2 specification.

C.1 Abstract Class `Caa`

```
;;;;;;;;;;;
Class Caa ;
;;;;;;;;;;;
Interface
  Type caaType;
  Create
    initCaa;
;;[...]
Body
  Use Exception, ListOfExcepts, ListOfObjects, ListOfRoles;
  Method
   createLocalCaaObj _ : objlist;
   startRoles _ _ _: rolelist rolelist objlist;
   acceptTest _ _ : rolelist outcome;
   endRoles _ _ : rolelist rolelist outcome;
   getExceptions _ _ : rolelist, exceptList;
   solve _ _ : exceptList exception;
   handleExcept _ _ : rolelist, exception;
  Transitions
    syncRoles;
    solveExceptions;
    executeEvt;
  Places
    Objs _ : objlist;
    Roles _ : rolelist;
    CalledRoles _ _ : roleType roleType;
  Axioms
;;            -------------------------------------------------
;;              GENERAL COORDINATION OF ROLES
;;            -------------------------------------------------

    syncRoles With
        Self.startRoles rlist clist caaObj ..
        Self.acceptTest rlist outc ..
        Self.endRoles rlist clist outc ::
      Objs caaObj, Roles rlist -> Objs caaObj, Roles rlist;
```

```
    startRoles [] [] CaaObj :: ->;
    startRoles (r ' rlist) (c ' clist) CaaObj With
        (c.callRole r arg .. r.start arg caaObj)
        // Self.startRoles rlist clist CaaObj:: ->;

    acceptTest (r ' []) outc With r.outcome outc:: ->;
    acceptTest (r ' rlist) outc1
        With (r.outcome outc1 // Self.acceptTest rlist outc2) ::
         (eq outc1 outc2) and (not empty? rlist) = true => ->;
    acceptTest (r ' rlist) abort
        With (r.outcome outc1 // Self.acceptTest rlist outc2) ::
         not(eq outc1 outc2) and not(eq outc1 fail)
          and not(eq outc2 fail) and not(empty? rlist) = true => ->;
    acceptTest (r ' rlist) fail
        With (r.outcome outc1 // Self.acceptTest rlist outc2) ::
         ((eq outc1 fail) or (eq outc2 fail))
           and (not empty? rlist) = true => ->  ;

    endRoles [] [] outc :: ->;
    endRoles (r ' rlist) (c ' clist) abort
        With (r.abort .. c.signalOutcome abort)
             // Self.endRoles rlist clist outc:: ->;
    endRoles (r ' rlist) (c ' clist) outc
        With c.signalOutcome outc // Self.endRoles rlist clist outc ::
        eq outc abort = false => ->;

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

    solveExceptions With
        Self.getExceptions rlist exlist ..
        Self.solve exlist e ..
        Self.handleExcept rlist e ::
          (eq e nil = false) and (e isIn? el) => Roles rlist -> Roles rlist;


    getExceptions [] [] :: ->;
    getExceptions (r 'rs) (e ' es)
        With r.exRaised e  //Self.getExceptions rs es :: ->;


    solve l fail :: fail isIn? l => ->;
    solve l abort :: not (fail isIn? l) and (abort isIn? l) => -> ;
;;  ** To be completed in the subclass according to the specific
;;     CAA's resolution graph. **
```

```
    handleExcept [] _ :: ->;
    handleExcept (r 'rs) e
        With r.handle e // Self.handleExcept rs e :: ->;
;;[...]
End Caa;
```

C.2 Abstract Class Role

```
;;;;;;;;;;;
Class Role ;
;;;;;;;;;;;
Interface
  Use Name, ListOfExpr, ListOfObjects, Outcome;
  Type roleType;
  Methods
     callRole _ _ : roleType objlist;
     outcome _ : exception;
     exRaised _ : exception;
     handle _ : exception;
     evtReq _ : event;
     objMgrAns _ : outcome;
  Create
    start _ _ : objlist objlist;
Body
  Use Instruction,
    Contxt, Substitution;
  Methods
     putExtObj _ : objlist;
     putIntObj _ : objlist;
     suspended? ;
     execute _ : event;
  Transitions
     eval ;
  Places
     RoleToCall _ : roleType objlist;
     ObjMgrReq _ : event;
     RaisedExcept _ : exception;
     Outcome _ : outcome;
     Instr _ : instruction;
     Instr' _ : instruction;
     Ctxt _ _ : varName expr; ;; local variables
     ExtObj _ : objType;     ;; external objects
```

```
IntObj _ : objType;     ;; internal objects
State _ : state;
```

Axioms

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

```
callRole r args :: RoleToCall r args -> ;
```

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

```
signalOutcome normal :: -> Instr End;
signalOutcome (ex! en exps) ::
   State terminating -> State terminating, Outcome (abort);
signalOutcome (ex! en exps) ::
    State executing, RaisedExcept nil ->
    State suspended, RaisedExcept (ex! en exps);

outcome outc :: Outcome outc -> ;
```

;; Collecting concurrently raised exceptions
;; --

```
exRaised e ::
   RaisedExcept e, State executing, Instr i ->
      RaisedExcept nil, State suspended;
exRaised e ::
   RaisedExcept e, State suspended, Instr i ->
      RaisedExcept nil, State suspended;
exRaised abort ::
   RaisedExcept e, State terminating, Instr i ->
      RaisedExcept nil, State suspended;
```

;; 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 abort ::
  State s, Outcome outc -> State terminating, Outcome
  abort;
```

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

;;  "Call  roleName( e1,.., en) Of  caaName"
     eval :: Instr (CallRole r withArgs args), Ctxt c,
             -> RoleToCall r (subst args c), Ctxt c;

;;  " If  c Then  instr1 Else  instr2"
     eval :: (subst cond c) = M-true =>
                Instr (If cond Then i1 Else i2), Ctxt c
                -> Instr i1, Ctxt c;
     eval :: (subst cond c) = M-false =>
                Instr (If cond Then i1 Else i2), Ctxt c
                -> Instr i2, Ctxt c;

;;   " instr1  ;  instr2"
     eval :: Instr (Sequence i1 i2), Instr' i3
             -> Instr i1, Instr' (Sequence i2 i3);
     eval :: (eq i iEnd) = false =>
                Instr iEnd, Instr' i -> Instr i, Instr' iEnd ;
     eval :: Instr iEnd Instr' iEnd -> Outcome normal;

;; [...]
End Role;
```