School of Computing Science,
University of Newcastle upon Tyne

Using the B Method for the Formalization of Coordinated Atomic Actions

Ferda Tartanoglu, Nicole Levy,
Valerie Issarny, Alexander Romanovsky

Technical Report Series

CS-TR-865

October 2004

Using the B Method for the Formalization of Coordinated Atomic Actions

Ferda Tartanoglu[1], Nicole Levy[2],
Valerie Issarny[1], and Alexander Romanovsky[3]

[1] INRIA Rocquencourt
78153 Le Chesnay, France
{Galip-Ferda.Tartanoglu, Valerie.Issarny}@inria.fr
http://www-rocq.inria.fr/arles/
[2] Laboratoire PRISM, Université de Versailles Saint-Quentin-en-Yvelines
78035 Versailles, France
Nicole.Levy@prism.uvsq.fr
[3] University of Newcastle upon Tyne
School of Computing Science, NE1 7RU, UK
Alexander.Romanovsky@newcastle.ac.uk

Abstract. Coordinated Atomic Actions have been proven successful for building dependable distributed systems due to their support for error recovery for both competitive and cooperative concurrent activities. This chapter introduces the formal specification of Coordinated Atomic Actions emphasizing the formalization of proposed dependability mechanisms using the B formal method. The specification then allows developing dependable systems, where the B formal specification can be refined to obtain a correct implementation of the associated runtime support.

1 Introduction

Dependability of systems is defined by the reliance that can be put on the service they deliver [5]. Developing distributed systems that are dependable is recognized as a complex task, requiring adequate mechanisms for dealing with the occurrence of failures. Coordinated Atomic Actions (CA Actions) [10] provide a general structuring mechanism for developing dependable systems through the exploitation of atomic actions and transactions. The composition of Coordinated Atomic Actions [7] further extends the base CA Action concept for developing open distributed systems.

Several applications have proven that Coordinated Atomic Actions are effective for building dependable concurrent systems [11, 2]. Dependability properties of such applications are proved using model-checking from their formal specifications using Petri nets and temporal logic [8, 6]. However, these specifications give formal verifications of dependability properties of specific applications and do not provide a general abstract formal model of Coordinated Atomic Actions. Such a model is given in [9] using timed CSP, however the specification is oriented towards the description of real-time safety-critical systems and the authors do not provide the formal specification of fault tolerance mechanisms of CA Actions, which is our main concern.

Our approach differs from the above work as it aims at providing a language for developing dependable distributed systems using fault tolerance mechanisms that are formally specified and implemented. Towards that goal, we provide a formal specification of the Coordinated Atomic Action model using the B formal method, from which we derive an XML-based language to be used to develop dependable distributed systems.

Properties of the Coordinated Atomic Action concept, and in particular the properties of the associated fault tolerance mechanisms (e.g., transactional accesses to shared external resources and embedded coordinated exception handling mechanisms) are specified in terms of invariants and pre-conditions on operations, which translate into run-time verification associated with the language's constructs. B refinement further allows offering an implementation of the run-time support associated with the language that is correct with respect to the B specification.

The chapter is structured as follows. Section 2 briefly presents Coordinated Atomic Actions and their composition. Sections 3 and 4 then introduce the B formal specification of the CA Action concept, discussing in particular the specification of fault tolerance mechanisms offered by CA Actions. Finally, Section 5 concludes, summarizing our contribution and discussing areas for future work.

2 Architecting Dependable Systems with Coordinated Atomic Actions

2.1 CA Actions

Coordinated Atomic Actions [10] (CA Actions) are a structuring mechanism for developing dependable concurrent systems through the generalization of the concepts of atomic actions [3] and transactions [4]. Atomic actions are used for controlling cooperative concurrency among a set of participating processes and for realizing coordinated forward error recovery using exception handling. Transactions are used for maintaining the coherency of shared external resources that are competitively accessed by concurrent actions. Each CA Action is designed as a multi-entry unit with roles activated by action participants, which cooperate within the action. A transaction is started on external objects and it terminates at the end of the CA Action.

A CA Action terminates with a normal outcome if no exception has been raised or if an exception has been raised and handled successfully; all transactions on external objects are then committed. If a participant raises an exception inside an action and if the exception cannot be handled locally by the participant, the exception is propagated to all the other participants of the CA Action for *coordinated error recovery*. If several exceptions have been raised concurrently they are resolved using a resolution tree imposing a partial order on all action exceptions, and the participants handle the resolved exception [3]. If coordinated recovery fails, the Coordinated Atomic Action terminates with an exceptional outcome. An exception is then signalled by the CA Action and transactions on external objects are aborted.

Coordinated Atomic Actions can be designed in a recursive way using action nesting. Several participants of a CA Action can co-enter into a *nested CA Action*, which

defines an atomic operation inside the embedding CA Action. Accesses to external objects within a nested action are performed as nested transactions so that if the embedding CA Action terminates exceptionally, all sub-transactions that were committed by nested actions are aborted as well. A CA Action participant can only enter one concurrent nested action at a time. Furthermore, a CA Action terminates only when all it's nested actions have completed. Note that if the nested action terminates exceptionally, an exception is signalled to the containing CA Action.

As an illustration, Figure 1 depicts a CA Action *A1* that is composed of three participants *P1, P2*, and *P3* and that comprises two nested CA Actions, *A11* and *A12*; nested transactions are further executed on external objects. An exception raised by participant *P2* is propagated to participants *P1* and *P2*, which causes the CA Action to enter an exceptional state, as shown by the greyed box, where the participants cooperate for handling the exception.

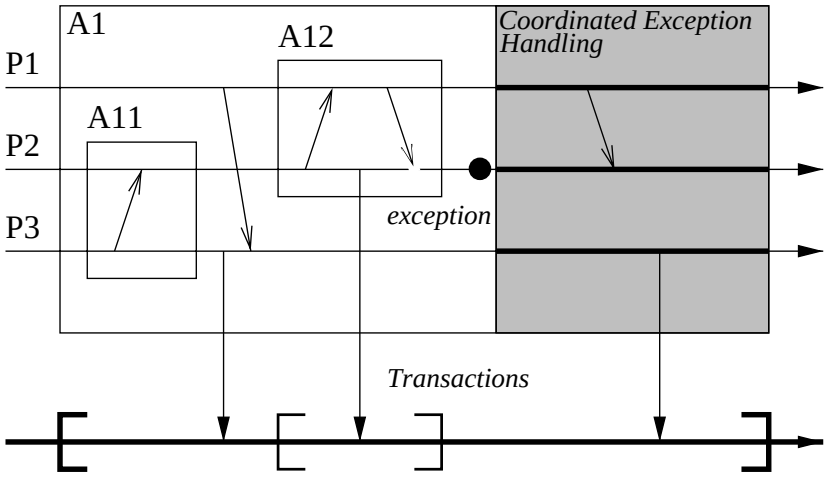

Fig. 1. Coordinated Atomic Actions

CA Actions mainly focus on structuring concurrent systems and on providing their fault tolerance by exception handling. One of the main intentions behind CA Actions is to employ them as a core mechanism for structuring complex distributed applications: they promote recursive view on system execution by abstracting away both normal and abnormal behaviour of the low level software.

2.2 CA Actions Composition

Composing Coordinated Atomic Actions allows the design of open distributed systems built out of several CA Actions [7]. Unlike base CA Action nesting where a subset of action participants co-enters into a nested action, *composed CA Actions* are autonomous

entities with their own participants and external objects. The resulting system is named a composite CA Action, built out of multiple autonomous composed CA Actions. In this model, any participant of a CA Action can dynamically initiate the creation of a composed CA Action. In addition, the model enables the reuse of existing CA Actions in different CA Action compositions.

The internal structure of a composed CA Action (i.e., set of participants, accessed external objects and participants' behaviour) is hidden from the calling CA Action, which only has access to the composed CA Action's interface. A participant that calls a composed CA Action enters a waiting state in a way similar to a synchronous RPC. The participant then resumes its execution according to the outcome of the composed CA Action. If the composed CA Action terminates exceptionally, its calling participant raises an internal exception which is possibly locally handled. If local handling is not possible, the exception is propagated to all the peer participants for coordinated error recovery. Note that unlike nesting, when a composed CA Action has terminated with a normal outcome, an abort operation of the containing CA Action does not automatically compensate effects of the composed one; specific handling must be performed at a higher level, e.g., a composed action can be initiated to abort or compensate actions on external objects if needed. We assume thus that transactions performed by composed CA Actions on external resources are compensable. It means that when a CA Action aborts and there is a composed CA Action that has committed before, then all transactions associated to this composed CA action have to be compensated.

Figure 2 illustrates the use of nested and composed CA Actions, considering a travel agency system. The top-level CA Action comprises the *User* and the *Travel* participants; the former interacts with the user while the latter achieves joint booking according to the user's request. The CA Action has further access to the *Banking System*. In a first step, the *User* participant requests the *Travel* participant to search for a trip. This leads the participants to enter the nested action *SearchTrip* in which the *Travel* participant invokes a composed action comprising the *Hotel* and the *Flight* participants. The external objects accessed by those participants are the hotel and flight booking system. The *SearchTrip* action, if successful, returns a list of possible trips. Then, according to the *User*'s selection, the *BookTrip* nested action is executed, leading to initiate another composed CA Action to book the given trip. If an exception is raised within the composed CA Action (e.g., *no_flight_available* for a given destination) and if it cannot be handled internally, the composed action terminates exceptionally by aborting all transactions on external objects and signals a failure exception to the higher level, i.e., to the calling participant.

3 Specifying Coordinated Atomic Actions in B

3.1 The B Method

B is a complete formal method [1] that supports a large part of the development life cycle, from abstract specification to implementation. The B formal method is a model-based method, which is based on set theory and predicate logic and extended by generalized substitutions. B specifications are represented by abstract machines encapsulating states (i.e., initialised variables and invariants) and state transformations described

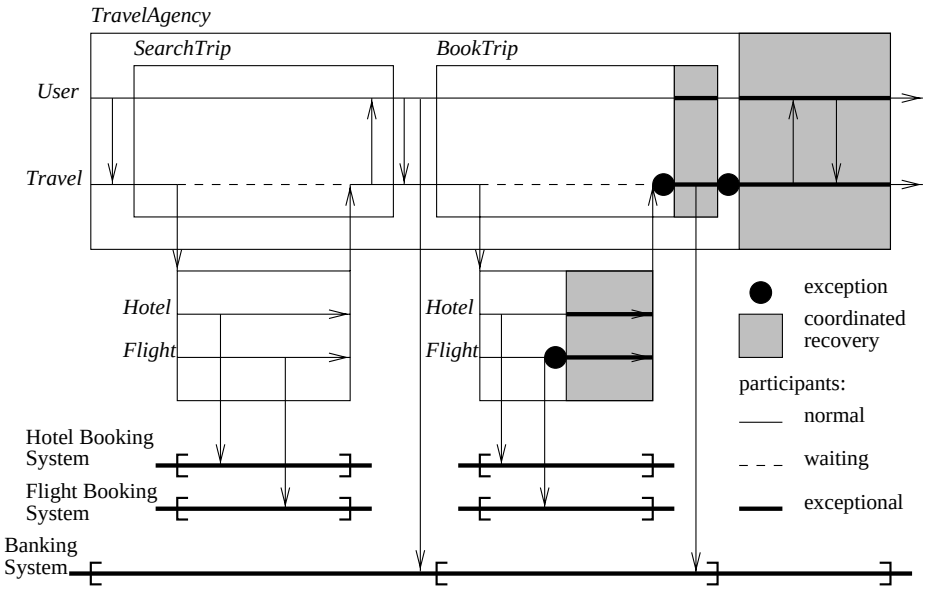

Fig. 2. CA Actions Composition

as operations (see Figure 3). Generally speaking, the B method allows us to define *abstract machines* and *refinements* over them in order to obtain equivalent but more concrete machines (see Figure 4). During the refinement, non-determinism is reduced and preconditions are relaxed, but the interfaces of the operations remain the same. At the end of the refinement process, an implementation can be written, which corresponds to an executable code.

Proofs are an essential part of the B formal method: it should be proven that all operations preserve the invariants of the machine, and that the implementations and refinements preserve the invariants and the behaviour of the initial abstract machine. There are various tools that help writing and proving B specifications. The main of them are B-Tool[4] and Atelier B[5]. Both tools include a type checker, an animator, a proof obligation generator, theorem provers, code translators and documentation facilities. We used the two software suites as we found in our investigation that they are complementary. We used thus a notation that is compatible with both of them.

3.2 Modelling Coordinated Atomic Actions

Our goal in developing the B specification of Coordinated Atomic Actions is to offer a general framework that can be instantiated to describe the implementation of a specific system that is developed using CA Actions. The framework defines the system compo-

[4] http://www.b-core.com/btool.html
[5] http://www.atelierb.societe.com

```
MACHINE
    M
VARIABLES
    v
INVARIANT
    i
INITIALISATION
    Init
OPERATIONS
    op=
      PRE
        P
      THEN
        S
      END;
END
```

Fig. 3. B Abstract Machine

$$M_0 \xrightarrow{refinement} M_1 \xrightarrow{refinement} \ldots \xrightarrow{refinement} M_n$$

Fig. 4. B Refinement Process

nents and associated fault tolerance mechanisms related to the dependability properties of CA Actions, which will be enforced for any system based on them.

For the sake of modularity, and for easing automated proofs, the B formal specification of Coordinated Atomic Actions is given by three abstract machines, each defining a set of the system components. These abstract machines are then composed using the B machine composition mechanisms as illustrated in Figure 5. In particular, the *EXTENDS* clause of B is used to compose several machines into one machine. Note that an additional abstract machine that defines some constants used by all machines is introduced and is included in all the definitions of the main abstract machines using the *SEES* clause, which makes possible to share resources provided by one machine with another.

The *PARTICIPANTS* abstract machine describes participants of a Coordinated Atomic Action and the *OBJECTS* abstract machine describes external objects that can be accessed by a CA Action participant during a transaction. The *CAACTIONS* abstract machine that extends the above two machines, defines operations associated with the execution of Coordinated Atomic Actions: creation, termination, nesting and composition of CA Actions, message exchange between participants, and exception handling. The *CONST* machine further contains global declarations and is seen by all the other machines.

In the remainder, we introduce the main elements of the B specification, focusing on dependability properties associated with CA Actions; the interested reader may find the complete specification at `http://www-rocq.inria.fr/~tartanog/dsos/`.

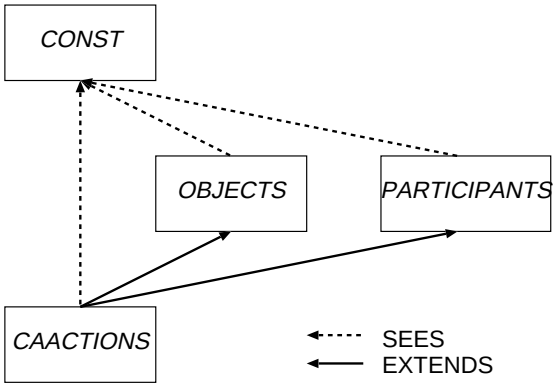

Fig. 5. Structure of the B Specification

3.3 Specifying States and State Transformations

The state of the overall system is defined by the states of the constituent abstract machines given by their variables and invariants. Operations are further defined and they describe the state transformations.

For instance, the state of the *PARTICIPANTS* abstract machine characterizes the participants of a Coordinated Atomic Action as follows:

– The *PARTICIPANT* set is declared in the *CONST* abstract machine and represents all possible participants that can be involved in a Coordinated Atomic Action. A participant that enters in a Coordinated Atomic Action is included in the subset *participant* of *PARTICIPANT* and removed at the end of the action:

$$participant \subseteq PARTICIPANT$$

– During the execution of a Coordinated Atomic Action, a participant's state can be normal, exceptional or waiting. The state is set to normal at the initialization of a CA Action. When an exception has been raised it is set to exceptional and the state is waiting if the participant invokes a composed Coordinated Atomic Action and is blocked until the action's termination. Since a participant can have different states in nested actions, we memorize them using a sequence:

$$PARTICIPANT\_STATE = \{normal, exceptional, waiting\}$$
$$participant\_state \in PARTICIPANT \rightarrow \mathbf{seq}(PARTICIPANT\_STATE)$$

– Each participant has a value (a set of variables), corresponding to the local state of the participant, that is initialised at the CA Action creation phase and logged for later use in case of backward recovery:

$$participant\_value \in PARTICIPANT \nrightarrow VALUE$$
$$initial\_values \in PARTICIPANT \nrightarrow \mathbf{seq}(VALUE)$$

The *OBJECTS* abstract machine defines the *OBJECT* set for all external objects that can be accessed by a participant. These objects are assumed to provide transactional interfaces to allow performing transactional operations such as *begin*, *commit*, and *abort*. If one or more participants access external objects, distributed transaction protocols, supported by the underlying systems of external objects, should be used. Furthermore, to allow nested CA Actions to access external shared resources, we require that the external objects implement some nested transaction protocol. In our specification we model transaction protocols in an abstract way (i.e., by defining the behaviours of main transactional operations *begin*, *commit* and *abort*). Further refinements of the abstract machine then would allow the specification of the aforementioned specialized protocols.

– The subset *object* specify external objects that take part in a transaction at a given time:

$$object \subseteq OBJECT$$

– The state of the external objects are defined using a variable for each object:

$$values \in OBJECT \rightarrow VALUE$$

The state of the *CAACTIONS* abstract machine, which specifies all CA Actions (top-level, nested and composed CA Actions) is defined by the following attributes.

– An abstract set *CAACTION* is introduced together with subset *caaction* of the CA Actions that are running at a given state of the system:

$$caaction \subseteq CAACTION$$

– Two types of actions are distinguished: top-level CA Actions and nested actions. In addition, two variables are used to memorize the nesting and composition relationships of Coordinated Atomic Actions:

$$is\_nested \in caaction \leftrightarrow caaction$$
$$is\_composed \in participant \nrightarrow caaction$$

– Coordinated Atomic Actions can be in a normal state (all the participants are in normal state) or exceptional (all the participants are in exceptional state):

$$CAACTION\_STATE = \{normal, exceptional\}$$
$$caaction\_state \in caaction \rightarrow CAACTION\_STATE$$

– Each CA Action has a set of participants and each of them participates to a sequence of nested CA Actions, so that in case of nesting, the nested CA Action is added to the sequence and is removed when the nested CA Action terminates:

$$participant\_of\_caaction \in caaction \rightarrow \mathcal{P}\ (participant)$$
$$caaction\_of\_participant \in participant \rightarrow \mathbf{seq}(caaction)$$

– CA Actions access several external objects:

$$caaction\_objects \in caaction \leftrightarrow object$$

Several invariant properties of the Coordinated Atomic Actions have been identified and specified. They are written as constraints on the variables of the abstract machines given above. Each abstract machine then defines the behaviour of the operations offered by CA Actions. Operations that are relevant to the fault tolerance mechanisms are discussed in the next section.

The operations that we have specified define several types of state transformations found in the CA Action concept such as the creation and termination of (possibly nested) atomic actions, the cooperation of participants, the CA Action composition processes, the transactional access of participants to externally shared resources, and the coordinated handling of exceptions:

– Operations on CA Actions:

- **create_main_caaction(*c, P, O*)** : Creates a main Coordinated Atomic Action *c*. A set of participants *P* are bound to the action created and a top-level transaction is initiated on the set of objects *O*.

- **create_nested_caaction(*c, c', P, O*)** : Creates a nested Coordinated Atomic Action *c* within the embedding action *c'*. A set of participants *P*, wich is a subset of the participants of *c'*, is bound to the nested action and a nested transaction is initiated on the set of objects *O*, a subset of external objects associated to *c'*;

- **compose_caaction(*p, c, P, O*)** : Called when participant *p* composes a new CA Action. The operation creates a new top-level Coordinated Atomic Action and sets the relationship composer-composed between the participant *p* that initiated the composition and the newly created atomic action *c*;

- **abort_caaction(*c*)**: Aborts the Coordinated Atomic Action *c*, sending a transactional abort message to all of its external objects. An alternative of this operation is **abort_nested_caaction** for aborting nested CA Actions;

- **terminate_caaction(*c*)**: Terminates the Coordinated Atomic Action *c*, either in a normal or in an exceptional state (defined as a separate operation **terminate_caaction_exceptional**), in which case an exception is raised in the containing action if any. Variants are **terminate_nested_caaction** and **terminate_nested_caaction_exceptional**;

– Operations on participants:

- {**send, recv**}**_message(*p, p', m*)** : Sends or receives a message *m* from one participant *p* to a peer participant *p'*;

- **raise_exception(*p, e*)**: Raises an exception *e* in participant *p*;

- **propagate_exception(*p*)**: Propagates the exception raised in *p* to all the participants of the last atomic action in which *p* takes part;

- **add_participant(*p*)**, **remove_participant(*p*)** and **set_participant_state(*p,s*)**: Operations called from other operations for adding and removing participants to (possibly nested) CA Actions and for setting their states.

– Operations on external objects:

- **add_objects(*O*)**: Called within a newly created top-level action. It initiates a transaction on external objects O. The variant operation **add_nested_objects(*O*)** is called from a nested CA Action to initiate nested transactions.

- **access_object(*p, o , f*)**: Applies function *f* to the external object *o* within a transaction identified by the CA Action in which the participant *p* takes part;

- **remove_objects(*O,v*)** and **remove_nested_objects(*O,v*)**: Terminate top-level or nested transactions on external objects O by either committing or aborting the transaction.

Invariant properties and the behaviours of the operations are defined focusing on dependability properties enforced by the fault tolerant mechanisms associated with CA Actions.

4 Enforcing Fault Tolerance

The fault tolerance mechanisms embedded within Coordinated Atomic Actions fall into three categories: (i) transactional access to shared external resources, (ii) atomicity and isolation of Coordinated Atomic Actions, and (iii) coordinated exception handling.

4.1 Transactions on External Objects

Accesses to external objects within Coordinated Atomic Actions are performed according to classical nested transaction rules.

– The operation that creates a top-level CA Action *c* (**create_main_caaction**) calls the operation **add_object(*caaction_objects[*{*c*}*]*)** that initiates the transaction on external objects associated to the CA Action given by the set *caaction_objects[*{*c*}*]* by affecting a *begin* value to these external objects:

add_objects(o) =
PRE
$o \subseteq OBJECT \wedge o \cap object = \emptyset$
THEN
$values := values \Leftarrow o \times \{begin\} \;||$
$object := object \cup o$
END;

Nested transactions are initiated using the similar **add_nested_object** operation. Preconditions on such operations ensures that general rules and constraints of the underlying transaction protocol are satisfied:

– Participants *P* of the nested CA Action *c* can only access a subset *O* of external objects associated to the containing CA Action *c'*. This constraint is ensured with the following precondition of the **create_nested_caaction(*c,c',P,O*)** operation:

$$\forall\, o.(o \in O \Rightarrow o \in caaction\_objects[\{c'\}])$$

– Then, the operation initiates a nested transaction on the external object using the (**add_nested_objects(*O*)**) operation call.

– When a (possibly nested) Coordinated Atomic Action terminates its execution normally (**terminate_caaction(*c*)** is called), it commits transactions on external objects within the **remove_objects(*O,v*)** operation:

remove_objects(caaction_ext_objects[{*c*}*],commit)*

– On the other hand, if the CA Action terminates exceptionally or aborts, all the transactions that it initiated on external objects are aborted as well:

$$remove\_objects(caaction\_ext\_objects[\{c\}],abort)$$

We have given in the above only interface operation calls on the underlying transactional supports of external objects (they are assumed to offer such a support as discussed in previous section). This supposes that nested transactions are aborted by the underlying transactional support of external objects.

4.2 Atomicity and Isolation of Coordinated Atomic Actions

Coordinated Atomic Actions are atomic in the sense that they support full backward error recovery in case of abortion (all transactions are aborted on external objects), i.e., they adhere to an *all-or-nothing* semantic. Coherency of shared resources is ensured using transactions as described above. Furthermore, all the execution of participants (e.g., cooperation of the participants) is encapsulated inside atomic and isolated computation units using nested and composed CA Actions. In the B specification, properties like atomicity and isolation are enforced using invariant properties, that preconditioned operations have to satisfy :

– The following invariant property of the *CAACTION* abstract machine states that participants of a nested CA Action *c* are also participants of the containing action *c'*:

$$\begin{aligned} &\forall\ (c, c').((c \in caaction \wedge c' \in caaction \wedge (c,c') \in is\_nested) \\ &\Rightarrow \quad participants\_of\_caaction(c) \\ &\qquad\qquad \subseteq participants\_of\_caaction(c')\ ) \end{aligned}$$

– In the case where a participant invokes the creation of a composed CA Action, participants *P* of the composed CA Action must not be involved in any other CA Action, which is simply ensured by the following precondition of the operation that initiates the composition **compose_caaction**:

$$P \cap participant = \emptyset$$

The isolation property of Coordinated Atomic Actions allows only intra-action communications. No information may be exchanged between actions unless a CA Action terminates and send results (or signal an exception) to a calling action. In the same way, sharing information using external objects follows rules of the underlying transaction model.

– Communication between participants *p* and *p'* within a CA Action (operations {**send, recv**}**_message(*p, p', m*)**) is realized by message exchanges. Preconditions of these operations set the rules of message exchange that is only allowed between participants of the same (possibly nested) CA Action. The participants must be in the same state (normal or exceptional). Finally, a participant that is in a waiting state (i.e., waiting for a composed CA Action to terminate) cannot send or receive any message:

$$caaction\_of\_participant(p) = caaction\_of\_participant(p') \wedge$$
$$\mathbf{last}(participant\_state(p)) = \mathbf{last}(participant\_state(p')) \wedge$$
$$\mathbf{last}(participant\_state(p)) \neq waiting$$

– Rules of nesting and composition are further specified with the following preconditions of the CA Action termination operations (**terminate_caaction(*c*)** and alternatives) stating, that a CA Action terminates when all embedded nested and composed CA Actions have terminated:

$$caaction \notin \mathbf{ran}(is\_nested) \wedge caaction \notin \mathbf{ran}(is\_composed)$$

– Furthermore, a participant can only enter one sibling nested CA Action at a time, which means that all participants in *P* willing to enter a nested CA Action are in the same containing CA Action. This constraint is enforced by the following precondittion of the operation **create_nested_caaction(*c, c', P, O*)**:

$$\mathbf{card}(\mathbf{ran}(\{\, p, c \mid p \in P \wedge c \in CAACTION \wedge$$
$$c = \mathbf{last}(caaction\_of\_participant(p)) \,\}\,)) = 1$$

– Finally, the participants *P* willing to enter a nested CA Action must all be in the same state, normal or exceptional. The precondition of the **create_nested_caaction** states:

$$\mathbf{card}(\mathbf{ran}(\{\, p, state \mid p \in P \wedge$$
$$state \in PARTICIPANT\_STATE \wedge$$
$$state = \mathbf{last}(participant\_state(p))\} \,)$$
$$) = 1 \wedge$$
$$\forall\, (p).(p \in P \Rightarrow \mathbf{last}(participant\_state(p)) \neq waiting\ )$$

4.3 Coordinated Exception Handling

Coordinated exception handling is started within a Coordinated Atomic Action when an exception raised by a participant has been propagated to all of the participants of the containing action. Each participant then resolves concurrent exceptions – if needed – and executes synchronously an exception handler for the exception resulting from the resolution. Rules for exception raising and propagation are specified by the following invariants and pre-conditioned operations.

– The following invariant of the *CAACTION* abstract machine ensures that a Coordinated Atomic Action is set to an exceptional state if all of its participants are in the exceptional state. Note that the participant can be in a waiting state following a call to a composed CA Action, in which case the test is performed on the last state of the participant before the call:

$$\begin{array}{l}\forall\,(c).\,(c \in caaction \wedge caaction\_state(c) = exceptional \\ \quad \Rightarrow \forall\,(p).(p \in participant\_of\_caaction(c) \\ \quad\quad \Rightarrow ((\mathbf{last}(participant\_state(p)) \in EXCEPTIONAL\_STATE) \vee \\ \quad\quad\quad ((\mathbf{last}(participant\_state(pa)) = waiting \wedge \\ \quad\quad\quad\; \mathbf{last}(\mathbf{front}(participant\_state(p))) \in EXCEPTIONAL\_STATE)))))\end{array}$$

Exception raising and propagation (to other participants) is realized by two operations defined in the *CAACTIONS* machine.

– The **raise_exception** operation requires that the participant and the CA Action are in the normal state, and sets the participant's state to exceptional. Note that when a coordinated exception handling fails and an exception should be signalled outside the action, we use the distinct operation **terminate_caaction_exceptional** :

$$\begin{array}{l}\mathbf{raise\_exception}\ (p, exception) = \\ \mathbf{PRE} \\ \quad p \in participant \wedge \\ \quad exception \in EXCEPTIONAL\_STATE \wedge \\ \quad \mathbf{last}(participant\_state(p)) = normal \wedge \\ \quad caaction\_state(\mathbf{last}(caaction\_of\_participant(p))) = normal \\ \mathbf{THEN} \\ \quad \mathbf{set\_participants\_state}(\{p\}, exception) \\ \mathbf{END};\end{array}$$

– The **propagate_exception** operation is then called to propagate the exception to all participants of the Coordinated Atomic Action. The *resolve_exception* function is used to resolve concurrently raised exceptions (the exact behaviour of this function will be defined in further refinements) and is defined as follows:

$$\begin{array}{l}resolve\_exception: (PARTICIPANT\_STATE \times \mathcal{P}(PARTICIPANT\_STATE)) \\ \qquad\qquad\qquad \rightarrow EXCEPTIONAL\_STATE\end{array}$$

The *propagate_exception* operation then requires that the CA Action where the exception will be propagated is in normal state (i.e., propagating an exception when the CA Action is in exceptional state is not allowed), and verifies that the participants are not engaged in any composed or nested CA Action. In that case, the pre-condition will not be satisfied and a further refinement may be for example (i) to delay the propagation until all embedded nested and composed action terminate

or (ii), to abort these actions and recover the state of the participants to the previous normal state. At the end of the propagation process that sets the participants states to exceptional, the Coordinated Atomic Action's state is set to exceptional as well:

propagate_exception(*p*) =

PRE

p ∈ *participant* ∧
last(*participant_state*(*p*)) ∈ *EXCEPTIONAL_STATE* ∧
caaction_state(**last**(*caaction_of_participant*(*p*))) = *normal* ∧
∀ *p'*.(*p'* ∈ *participant_of_caaction*(**last**(*caaction_of_particip*(*p*)))
⇒ **last**(*participant_state*(*p'*)) ≠ *waiting*) ∧
card(**ran**({*p'*,*c* |
p' ∈ *participant_of_caaction*(
last(*caaction_of_participant*(*p*))) ∧
c ∈ *CAACTION* ∧ *c* = **last**(*caaction_of_participant*(*p'*))
})) = 1

THEN

LET *ps* **BE**
ps = { *p'*, *s* |
p' ∈ *participant_of_caaction*(
last(*caaction_of_participant*(*p*))) ∧
s ∈ *PARTICIPANT_STATE* ∧
s= **last**(*participant_state*(*p'*))
}
IN
LET *s'* **BE**
s' = *resolve_exception*(**last**(*participantant_state*(*p*)),**ran**(*ps*))
IN
set_participants_state (*participant_of_caaction*(
last(*caaction_of_participant*(*p*))), *s'*) ||
caaction_state(**last**(*caaction_of_participant*(*p*))) := *exceptional*
END
END

END;

If a Coordinated Atomic Action terminates in an exceptional state, all transactions on external objects are aborted by calling the operation **remove_objects(*O, abort*)** within the operation **terminate_caaction_exceptional(*c*)**. If the CA Action is a nested or a composed one, then an exception is *signalled* to a higher level.

5 Conclusion

This chapter has presented how to specify fault tolerance mechanisms using the B formal method. We have considered the use of Coordinated Atomic Actions that have been proved useful for building dependable systems. We have defined a generic formal specification using the B method, defining systems composed of several Coordinated Atomic Actions that make concurrent accesses to external objects. B was chosen because of its powerful theorem proving ability and because of availability of a number of mature tools. We have shown how to specify the following dependability mechanisms of CA Actions: (i) constraints related to the atomic accesses to external transactional objects, (ii) encapsulation of computations inside atomic action units ensured through action nesting and composition and (iii), properties related to the behaviour of the system in case of exception occurrences.

In order to have an implementation of the CA Action's run-time support, the abstract machines should be refined. At the end of the refinement process, we will have an executable code that correspond to the implementation of the operations defining the B machines, offered as a programming library. Note that when implementing the CA Actions, some existing libraries such as drivers for running transactions on external shared resources may be used. For all these libraries, what is usually known is the interfaces of the offered methods. In order to be able to prove the correctness of the implementation it would be necessary: (i) to have in addition the formal specification of the behaviour of these methods and (ii), to prove that the refinements of the machines that use these methods are correct (in the B sense). During the refinement, the non-determinism will be reduced (e.g., by introducing of reliable message queues if sending a message is not possible when the receiving participant is in a waiting state, resulting in a false pre-condition). The preconditions have to be relaxed in order to take into account all the possible cases. The formal specification together with the refinement process give an executable code that is correct with respect to the specification.

Up to now several implementations of Coordinated Atomic Actions have been proposed and experimented with, but mainly on closed systems [11, 2]. We are working on an implementation of CA Action-based systems to be defined as a composition of Web services such as a travel agency that composes several existing autonomous Web services. This kind of loosely-coupled system compositions clearly needs new dependability properties (e.g., relaxed atomicity properties for accessing external objects). We intend to use this initial B specification to define and to study such properties.

Acknowledgments

This research was partially supported by the European IST DSoS (Dependable Systems of Systems) project (IST-1999-11585)[6] and by the *ACI Sécurité Informatique* project COrSS (*Composition et raffinement de systèmes sûrs*)[7].

[6] `http://www.newcastle.research.ec.org/dsos/`
[7] `http://www.irit.fr/CORSS/`

References

1. J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.
2. D.M. Beder, A. Romanovsky, B. Randell, C.R. Snow, and R.J. Stroud. An Application of Fault Tolerance Patterns and Coordinated Atomic Actions to a Problem in Railway Scheduling. *ACM, Operating Systems Review*, 34(4):21–31, October 2000.
3. R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, SE-12(8), 1986.
4. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
5. J.-C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Wien, New York, 1992.
6. D. Schwier, F. von Henke, J. Xu, R.J. Stroud, A. Romanovsky, and B. Randell. Formalization of the CA Action Concept Based on Temporal Logic. Design for validation (deva) basic esprit project. second year report. part 2, LAAS, France, 1997.
7. F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Coordinated Forward Error Recovery for Composite Web Services. In *Proceedings of the* 22^{nd} *IEEE Symposium on Reliable Distributed Systems*, pages 167–176, Florence, Italy, October 2003.
8. J. Vachon, N. Guelfi, and A. Romanovsky. Using COALA to Develop a Distributed Object-Based Application. In *Proceeding of the 2nd Int. Symposium on Distributed Objects and applications (DAO'00)*, pages 195–208, Antverp, Belgium, 2000.
9. S. Veloudis and N. Nissanke. Modelling Coordinated Atomic Actions in Timed CSP. In *Proceedings, 6th International Symposium on Formal Techniques in Real-Time Fault Tolerant Systems: FTRTFT 2000. Pune, India.*, volume 1926 of *LNCS*, pages 228–239. Springer Verlag, 2000.
10. J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the Twenty-Fifth IEEE International Symposium on Fault-Tolerant Computing*, 1995.
11. J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A.F. Zorzo, E. Canver, and F. von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Transactions on Computers*, 51(2):164–179, 2002.