

Chapter 6

TLA SPECIFICATION OF A MECHANISM FOR CONCURRENT EXCEPTION HANDLING

Avelino Francisco Zorzo

Faculdade de Informática - PUCRS - 90619-900 - Porto Alegre - RS - Brazil
zorzo@inf.pucrs.br

Brian Randell

University of Newcastle upon Tyne - NE1 RU - Newcastle upon Tyne - UK
brian.randell@ncl.ac.uk

Alexander Romanovsky

University of Newcastle upon Tyne - NE1 RU - Newcastle upon Tyne - UK
alexander.romanovsky@ncl.ac.uk

Abstract Recently the concept of *dependable multiparty interaction* (DMI) has been introduced. In a multiparty interaction, several parties (objects or processes) somehow “come together” to produce an intermediate and temporary combined state, use this state to execute some activity, and then leave this interaction and continue their normal execution. The concept of multiparty interactions has been investigated by several researchers, but to the best of our knowledge none have considered how failures in one or more participants of the multiparty interaction could be dealt with. In this paper, we show how this mechanism deals with concurrent exceptions raised during an interaction. This is shown through a formal description of the DMI concept. We use Temporal Logic of Actions (TLA) in order to formally describe the DMI features.

Keywords: Distributed and Parallel Systems, Multiparty Interactions, Concurrent Exception Handling

P. Ezhilchelvan and A. Romanovsky (eds.) *Concurrency in Dependable Computing*.
© 2002 Kluwer Academic Publishers. Printed in the Netherlands.

1. Introduction

It is very common for software developers to write programs under the optimistic assumption that nothing will go wrong when the program is executed. Unfortunately, there are many factors that can make this assumption invalid. For example, an arithmetic expression that may cause a division by zero; an array that is indexed with a value that exceeds the declared bounds; the square root of a negative number; a request for memory allocation during run-time that may exceed the amount of memory available; opening a file that does not exist; and many more.

When any of such event happens the system will often fail in an unexpected way. This is not acceptable in current programming standards. To improve reliability, it is important that such circumstances are detected and treated appropriately. Conventional control structures, such as the if-then-else command, are inadequate. For example, to check that an index of an array is always valid, a programmer could explicitly test the value of the index each time before using it, which is cumbersome and could often be forgotten or intentionally omitted. A better way would be to rely on the underlying system to trap the situation where array indexes are outside the array bounds. To cope with this kind of situation, several programming languages provide features for handling such circumstances, i.e. exception handling.

Exception handling in sequential programs is a well-known subject with several languages providing mechanisms for handling exceptions. Exception handling in parallel programs is much more complex than in sequential programs. Exceptional termination in a process can have a strong impact on other processes. For example, consider a set of processes that communicate with each other via a rendezvous mechanism. A process may terminate abruptly due to the presence of an exception. Processes that need to communicate with the process that was terminated, may be suspended for ever because the terminated process will not be ready for communication anymore.

The problem of dealing with concurrent exceptions has been addressed in different systems [1] and models [2] [3]. For example, the VAXELN programming environment from Digital [1] provides means for a process to raise exceptions in other processes. The raising of an exception in a different process is done in an unstructured manner. A process can enable or disable this kind of exceptions. Raising an exception in a process that has disabled this kind of exception has no effect. The approach of allowing an exception in a process to be raised in a different process outside a structured framework can have a devastating effect

on program modularity. This is especially the case when the raising of exceptions in other processes cannot be restricted.

In [2], a model for dealing with concurrent exceptions explores the use of an exception tree. Exceptions that can be signalled by a component of a parallel block C are organized in a tree structure. The root of this tree contains the *universal exception*, i.e. the exception that represents the whole exceptional domain of C. When more than one exception is raised concurrently, a handler for an exception that is ancestor to all exceptions raised is executed. In the worst case scenario, a handler for the *universal exception* is executed. In [3], a different model that relies on the definition of resolution functions within classes is presented. In this model, a resolution function takes a sequence of exceptions as input parameter and returns an exception.

Despite the aforementioned efforts, mechanisms for dealing with concurrent exceptions in programming languages are still in their early stages. For example, in the Ada 95 [4] rendezvous mechanism, if an exception is raised during a rendezvous and not handled in the **accept** statement, that exception is propagated to both tasks and must be handled in two places. However, Ada 95 does not provide any mechanism to handle concurrent exceptions.

A mechanism that handles concurrent exception has been presented in [5]. This mechanism is called *dependable multiparty interaction* (DMI) and is able to cope with several concurrent exceptions being raised during an activity executed jointly by a set of participants (processes/threads).

The main goal of this paper is to present how Temporal Logic of Actions (TLA) [6] can be applied to describe dependable programming/specification constructs, such as the DMI concept. TLA is a formalism suitable for describing state transition systems and their properties using a uniform notation. Hence being suitable to describe fault tolerance mechanisms. For example, the one described in this paper.

This paper is organized as follow. Section 2 introduces the concept of dependable multiparty interaction. Section 3 presents the formalism used to describe the DMI properties. Section 4 shows the formal description of the DMI properties. Section 5 discusses the related work. Section 6 draws some conclusions.

2. Dependable Multiparty Interactions

Existing multiparty interaction mechanisms [7] [8] do not provide features for dealing with possible failures that may happen during the execution of the interaction. Typically, the underlying system that is executing those multiparty interactions will simply stop the system in

response to a failure. In DisCo [9], for instance, if an assertion inside an action is false, then the run-time system is assumed to stop the whole application. This is unacceptable in many situations, e.g. a flying aircraft or a pacemaker (see [10] for more examples).

In this section, multiparty interactions are augmented with an exception handling facility to form a new construct: the *dependable multiparty interaction* (DMI). The DMI mechanism is based on a structuring mechanism called Coordinated Atomic Action (CA action) [11], which brings together the concept of conversation [12] and transaction [13]. Specifically, a DMI provides facilities for:

- **HANDLING CONCURRENT EXCEPTIONS:** when an exception occurs in one of the bodies of a participant, and is not dealt with by that participant, the exception must be propagated to all participants of the interaction [2]. A DMI must also provide a way of dealing with exceptions that can be raised by one or more participants. Finally, if several different exceptions are raised concurrently, the DMI mechanism has to decide which exception will be raised in all participants.

With respect to how the participants of a DMI will be involved in the exception resolution and exception handling, there are two possible schemes: synchronous or asynchronous. In synchronous schemes, each participant has to either come to the action end or to raise an exception; it is only afterwards that it is ready to participate in any kind of exception handling; this means that the participant's execution cannot be pre-empted if another participant raises an exception. In asynchronous schemes, participants do not wait until they finish their execution or raise an exception to participate in the exception handling; once an exception is raised in any participant of the DMI, all other participants are interrupted and handle the raised exceptions together. Although implementing synchronous schemes is easier than asynchronous, because all participants are ready to execute the exception handling, the synchronous scheme can bring the undesirable risk of deadlock. Therefore the asynchronous scheme is adopted;

- **ASSURING CONSISTENCY UPON EXIT:** participants can only leave the interaction when all of them have finished their roles and the external objects are in a consistent state. This property guarantees that if something goes wrong in the activity executed by one of the participants, then all participants have an opportunity to recover from possible errors.

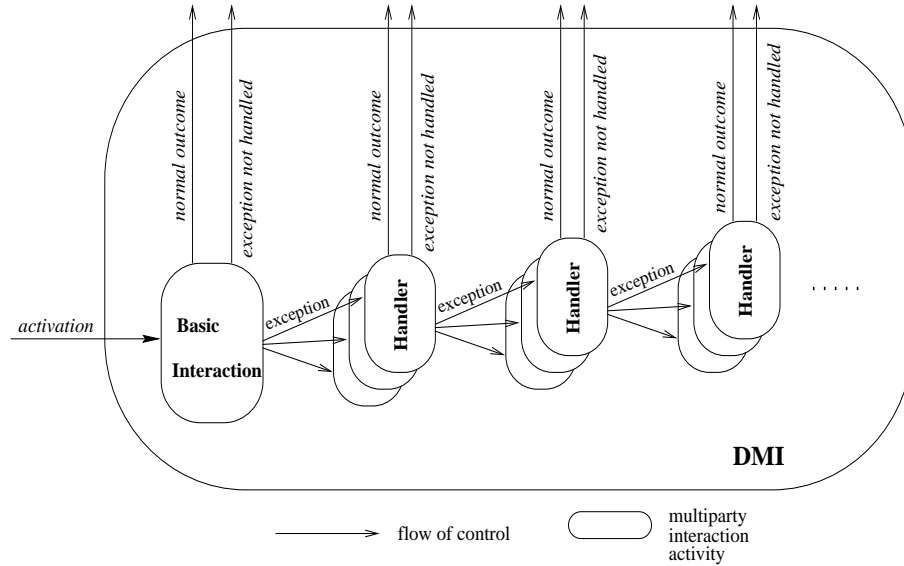


Figure 1. Dependable Multiparty Interaction

The key idea for handling exceptions is to build DMIs out of not necessarily reliable multiparty interactions by chaining them together, where each multiparty interaction in the chain is the exception handler for the previous multiparty interaction in the chain. Figure 1 shows how a basic multiparty interaction and exception handling multiparty interactions are chained together to form a composite multiparty interaction, in fact what we term a DMI, by handling possible exceptions that are raised during the execution of the DMI. As shown in the figure, the basic multiparty interaction can terminate normally, raise exceptions that are handled by exception handling multiparty interactions, or raise exceptions that are not handled in the DMI. If the basic multiparty interaction terminates normally, the control flow is passed to the callers of the DMI. If an exception is raised, then there are two possible execution paths to be followed: *i*) if there is an exception handling multiparty interaction to handle this exception, then it is activated by all roles in the DMI; *ii*) if there is no exception handling multiparty interaction to handle the raised exception, then this exception is signalled to the invokers of the DMI. The whole set of basic multiparty interaction and their associated exception handling multiparty interactions form a single entity: they are isolated from the outside so that internal activities (e.g., the raising of an exception) are not visible to the enclosing environment.

The exceptions that are raised by the basic multiparty interaction or by a handler, should be the same for all roles in the DMI. If several roles raise different concurrent exceptions, the DMI mechanism activates an exception resolution algorithm based on [2] to decide which common exception will be raised and handled.

In view of our interest in dependability, and in particular fault tolerance, we adopt the use of pre and post-conditions, which are checked at run-time. Regarding the remaining alternatives presented in [7] and [14], we have made the following design choices for DMIs:

- although the particular processes involved should be able to vary from one invocation of a DMI to the next, their number in a given DMI should be fixed;
- the processes should synchronise their entry to and exit from the DMI;
- the DMI mechanism should ensure that as viewed from outside the DMI, its system state should change atomically, though inside the DMI intermediate internal states will be visible;
- the way the underlying system executes a DMI can be synchronous or asynchronous.

The choice for allowing a varying set of processes to enrol into a DMI is related to the expressive power of the language construct we intend to provide. In [7] a taxonomy of languages that provide multiparty interactions as a basic construct is presented. In the presented taxonomy, the basic construct that presents the higher degree of expressiveness is a team. A DMI is a team, hence choice (i) was made. Synchronisation upon entry and exit (choice (ii)) is crucial if we want to have some kind of guard to be tested before the DMI commences, or an assertion to be tested before the DMI terminates. For example, if participants in a DMI are allowed to terminate without synchronising upon exit, then the process of involving that participant in the handling of an exception raised by another participant of the DMI will be much more difficult. Paper [15] discusses several issues related to termination of processes that should not interfere with each other, e.g. issues related to error recovery before a process has terminated, or error recovery after a process has terminated the execution of an activity. Choice (iii) is related to the visibility of shared data inside the DMI and outside of the DMI. The related “frozen initial state” property discussed in [14] is used in relation to the participants that are outside the DMI, i.e. they see the change of shared data as being instantaneous when the DMI terminates. Our

proposal differs from [14] in relation to the visibility of shared data inside the DMI. In our proposal, participants can exchange data inside the DMI, while in [14] participants of a multiparty interaction view shared data as “frozen” when the multiparty interaction commences.

3. Formal Semantics

A well-defined syntactic and semantic description of a language is essential for helping good design and programming of a system. The *syntax* of a language describes the correct form in which programs can be written while the *semantics* expresses the meaning that is attached to the various syntactic constructs. While *syntax diagrams* and *Backus-Naur Form - BNF* have become standard tools for describing the syntax of a language, no such tools have become widely accepted and standard for describing the semantics of a language. Different formal approaches to semantics definition exist, e.g. *operational semantics*, *axiomatic semantics*, or *denotational semantics*. Several authors report how to use these approaches for describing the semantics of programming languages [16] [17].

Concurrent systems are usually described in terms of their behaviour - what they do in the course of an execution [18]. The Temporal Logic formal model [19] was introduced to describe such behaviour of concurrent systems. A variation of Temporal Logic that makes it practical to write a specification as a single formula was presented in [6]. This variation is called Temporal Logic of Actions - TLA. TLA provides the mathematical basis for describing properties of concurrent systems.

3.1 Temporal Logic of Actions

The Temporal Logic of Actions - TLA [6] is a formalism suitable for describing state transition systems and properties of such systems using the same notation.

TLA is a linear-time logic in which expressions are evaluated for non-terminating sequences of states. Each sequence of states is called a behaviour. A state is an assignment of values to variables. Variables that are used to model properties are state functions, which have unique values in each state. A state function is a non-boolean expression built from variables, constants, and constant operators. Semantically, a state function assigns a value to each state. An individual state change is called a step. A step that allows variables to stay unchanged is called a stuttering step.

An action is a boolean expression containing primed and unprimed variables. For any pair of states, primed variables refer to the second

```

process example is
  integer x := 0;
  body
    loop x := x + 1; end;
  end body
end process

```

Figure 2. Simple Program in DIP

state whereas unprimed ones refer to the first state. An action is said to be enabled in a state s if and only if there exists some state t such that the pair of states $\langle s, t \rangle$ satisfies that action.

Rather than presenting the full description of TLA, a simple program [6] in Dependable Interacting Processes (DIP) [20] is presented with its corresponding TLA formula. The process, in Figure 2, initialises a variable x with 0 and then keeps incrementing x by 1 forever.

The TLA formula for the above DIP process is defined as follows:

$$\begin{aligned} \square &\triangleq \wedge (x = 0) \\ &\wedge \square [x' = x + 1]_x \\ &\wedge \text{WF}_x(x' = x + 1) \end{aligned}$$

A TLA formula is true or false on a behaviour. Formula \square , presented above, is true on a behaviour in which the i^{th} state assigns the value $i - 1$ to x , for $i = 1, 2, \dots$. In the above TLA formula, the conjunct $(x = 0)$ specifies that initially, x is equal to 0; the conjunct $\square [x' = x + 1]_x$ specifies that the value of x in the next state (x') is always (\square) equal to its value in the current state (x) plus 1. The subscript x specifies that stuttering steps are allowed, i.e. steps where the value of x is left unchanged. The $\text{WF}_x(x' = x + 1)$ conjunct rules out behaviours in which x is incremented only a finite number of times. It asserts that, if the action $(x' = x + 1) \wedge (x' \neq x)$ ever becomes enabled and remains enabled forever, then infinitely many $(x' = x + 1) \wedge (x' \neq x)$ steps occur. WF stands for Weak Fairness.

In the next section the specification of the dependable multiparty interaction mechanism is presented in TLA.

4. DMI in TLA

In this section we will present the semantics of dependable multiparty interactions. However, before we start formally describing the semantics of DMI in TLA, consider the following:

- a DMI is represented by a set of roles that are executed by players;
- a player has to activate a role in DMI in order to execute the commands inside a role;
- a DMI only starts when all roles of the DMI have been activated, and the guard (boolean expression) at the beginning of the DMI is true;
- the DMI only finishes when all players have finished executing their roles, and the assertion at the end of the DMI (boolean expression) is true (if no exceptions were raised);
- roles can only access data that is sent to them when they are activated, or data that is sent to them by other roles belonging to the same DMI;
- exceptions may be raised during the execution of a DMI, in which case all roles that have not raised an exception are interrupted; an exception resolution algorithm is executed when all roles either have raised an exception or have been interrupted.
- if there is a handler to deal with the exception that was decided upon by the exception resolution algorithm, then this handler is activated by all roles;
- if there is no handler to deal with the exception that was decided upon by the exception resolution algorithm, then the exception is raised in the callers of all roles;
- handlers have the same number of roles as the DMI to which they are connected.

In order to formally specify the semantics of a DMI in TLA, we will use the following sets, predicates and state variables:

- **Exceptions:** the set of exceptions handled by the DMI;
- **Commands:** a set of commands;
- **Objects:** a set of objects;

- **Players**: a set of players that can participate in a DMI;
- **Roles**: contains the roles of a DMI. Each element of this set is a record with a field to represent the `state` of the role, a field to represent the `result` of the role after the commands of this role have been executed, a field to store those `commands`, and a field containing the set of `objects` manipulated by the role;
- **Handlers**: the set of handlers for the DMI;
- **GuardExpression(e)**: a predicate representing the execution of the precondition of the DMI. The parameter `e` contains the set of all tuples $\langle p, er, o \rangle$, where `p` represents a player that is enroled to the role `er`, and `o` is the set of objects sent to the role by the player;
- **AssertionExpression(e)**: the same as **GuardExpression(e)** but for the post-condition of the DMI;
- **ExecuteCommands(e)**: execute the commands for the corresponding role;
- **Resolve(enroled)**: execute the exception resolution algorithm for all roles in the DMI. After this algorithm has been executed all roles will produce the same exceptional `result`;
- four state variables: *i*) `guard`, which indicates whether the DMI can be started or not; *ii*) `assert`, which indicates whether the DMI was finished successfully or not; *iii*) `enroled`, which stores the roles that have already been enroled to in a particular execution of the DMI; and, *iv*) `elements`, which stores the tuples $\langle p, er, o \rangle$ that are used when executing the roles commands.

The type invariant specifies that the `guard` and `assert` are `BOOLEAN` variables, the state of a role can only have one of the values from the set `{"wait", "ended", "started"}`, and the result of a role can either have a value from the set `{"ok", "interrupted"}` or from the set of possible exceptions in `Exceptions`. The type invariant is defined as:

$$\begin{aligned}
 \text{TypeInvariant} &\triangleq \wedge \text{guard, assert} \in \text{BOOLEAN} \\
 &\wedge \forall r \in \text{Roles}: \\
 &\quad r.\text{state} \in \{\text{"wait"}, \text{"ended"}, \text{"started"}\} \\
 &\wedge \forall r \in \text{Roles}: \\
 &\quad r.\text{result} \in \{\text{"ok"}, \text{"interrupted"}\} \cup \text{Exceptions} \\
 &\wedge \text{enroled} \subseteq \text{Roles} \\
 &\wedge \text{elements} \subseteq \text{Players} \times \text{Roles} \times \text{Objects}
 \end{aligned}$$

The initial condition for the DMI is that all roles are in a waiting state, both `guard` and `assert` have the value `FALSE`, and the `enroled` and `elements` sets are empty. The `Init` predicate is defined in TLA as:

$$\begin{aligned} \text{Init} \triangleq & \wedge \forall r \in \text{Roles}: r.\text{state} = \text{"wait"} \\ & \wedge \text{guard} = \text{FALSE} \\ & \wedge \text{assert} = \text{FALSE} \\ & \wedge \text{enroled} = \{\} \\ & \wedge \text{elements} = \{\} \end{aligned}$$

For a player `p` to enrole in a role `er` with a set of objects `o`, it has to execute the action `Enrole(p,er,o)`. This step is only enabled if role `er` belongs to the set of `Roles` in the DMI and no other player has enroled to such a role. This is expressed by the first two conjuncts of the following TLA formula. If this step is enabled, then the role `er` is added to the `enroled` set and the tuple `<p, r, o>` is added to the `elements` set. The `Enrole(p,er,o)` action is defined in TLA as:

$$\begin{aligned} \text{Enrole}(p,er,o) \triangleq & \wedge er \in \text{Roles} \\ & \wedge er \notin \text{enroled} \\ & \wedge \text{enroled}' = \text{enroled} \cup \{er\} \\ & \wedge \text{elements}' = \text{elements} \cup \{<p,er,o>\} \\ & \wedge \text{UNCHANGED } \langle \text{guard}, \text{assert} \rangle \end{aligned}$$

The DMI only begins if all roles have a player enroled to and the precondition is true. The testing of the guard with all players enroled is defined by the following two TLA conjunctions:

$$\begin{aligned} \text{Guard} \triangleq & \wedge \forall r \in \text{Roles}: r \in \text{enroled} \\ & \wedge \text{guard}' = \text{GuardExpression}(\text{elements}) \\ & \wedge \text{UNCHANGED } \langle \text{enroled}, \text{elements}, \text{assert} \rangle \\ \text{Begin} \triangleq & \wedge \text{guard} = \text{TRUE} \\ & \wedge \forall r \in \text{Roles}: r.\text{state}' = \text{"started"} \\ & \wedge \text{UNCHANGED } \langle \text{enroled}, \text{elements}, \text{assert}, \text{guard} \rangle \end{aligned}$$

The execution of all roles is defined in the action `ExecuteRoles`. This step is only enabled if all roles have `state = "started"`. If enabled, then the result of the execution of the set of commands of a role is stored in the field `result`. The `ExecuteRoles` is defined in TLA as:

$$\begin{aligned} \text{ExecuteRoles} \triangleq & \wedge \forall r \in \text{Roles}: r.\text{state} = \text{"started"} \\ & \wedge \forall \langle p,r,o \rangle \in \text{elements}: \\ & \quad r.\text{result}' = \text{ExecuteCommands}(\langle p,r,o \rangle) \\ & \wedge \text{UNCHANGED } \langle \text{enroled}, \text{assert}, \text{guard} \rangle \end{aligned}$$

When all roles have executed their commands without raising an exception, i.e. their state is equal to `"ok"`, the post-condition expression

can be tested. The `assert` variable changes its value based on the execution of the `AssertionExpression(elements)` action. The post-condition of a DMI is defined as:

$$\begin{aligned} \text{Assertion} \triangleq & \wedge \forall r \in \text{Roles}: r.\text{result} = \text{"ok"} \\ & \wedge \text{assert}' = \text{AssertionExpression}(\text{elements}) \\ & \wedge \text{UNCHANGED } \langle \text{enroled}, \text{elements}, \text{guard} \rangle \end{aligned}$$

If no exceptions were raised, then the normal termination of a DMI is defined in the `NormalEnd` action. The condition that enables this step is `assert = TRUE`, i.e. the post-condition was passed. This step changes the state of all roles to "wait", meaning that the roles are ready to be executed again. The sets `enroled` and `elements` are emptied. The TLA definition of `NormalEnd` is:

$$\begin{aligned} \text{NormalEnd} \triangleq & \wedge \text{assert} = \text{TRUE} \\ & \wedge \forall r \in \text{Roles}: r.\text{state}' = \text{"wait"} \\ & \wedge \text{enroled}' = \langle \rangle \\ & \wedge \text{elements}' = \langle \rangle \\ & \wedge \text{assert}' = \text{FALSE} \\ & \wedge \text{guard}' = \text{FALSE} \end{aligned}$$

Figure 3 shows the complete first part of the formal semantics of a DMI. In the figure all conjunctions are related to the normal execution of a DMI. In Figure 4, we define the formal semantics for the steps that are taken in case of one or more exceptions being raised. An exception can be raised during the execution of the set of commands of a role in the `ExecuteCommands` action.

The activation of a handler depends on the state of the roles. A handler is only activated when all roles have the same value for their `result`, and there exists a handler for the exception resolved by the resolution algorithm. The activation of a handler is defined as:

$$\begin{aligned} \text{ActivateHandler} \triangleq & \wedge \forall r_1, r_2 \in \text{Roles}: (r_1.\text{result} = r_2.\text{result}) \\ & \wedge \exists h \in \text{Handlers}: (\exists r \in \text{Roles}: r.\text{result} \in h.\text{Exc}) \\ & \wedge \text{UNCHANGED } \langle \text{enroled}, \text{elements}, \text{assert}, \text{guard} \rangle \end{aligned}$$

The resolution algorithm on the other hand, is activated once all roles have raised an exception, i.e. their `result` belongs to the set `Exceptions`, or have been interrupted. The state of all roles has to be different from "ok". This action is defined as:

$$\begin{aligned} \text{ExceptionResolution} \triangleq & \wedge \forall r \in \text{Roles}: r.\text{result} \neq \text{"ok"} \\ & \wedge \text{Resolve}(\text{enroled}) \\ & \wedge \text{UNCHANGED } \langle \text{enroled}, \text{elements}, \text{assert}, \text{guard} \rangle \end{aligned}$$

MODULE DMI

EXTENDS Naturals, Sequences
 VARIABLES enroled, elements, guard, assert

Init \triangleq $\wedge \forall r \in \text{Roles} : r.\text{state} = \text{"wait"}$
 $\wedge \text{guard} = \text{FALSE}$
 $\wedge \text{assert} = \text{FALSE}$
 $\wedge \text{enroled} = \{\}$
 $\wedge \text{elements} = \{\}$

TypeInvariant \triangleq $\wedge \text{guard}, \text{assert} \in \text{BOOLEAN}$
 $\wedge \forall r \in \text{Roles} : r.\text{state} \in \{\text{"wait"}, \text{"ended"}, \text{"started"}\}$
 $\wedge \forall r \in \text{Roles} : r.\text{result} \in \{\text{"ok"}, \text{"interrupted"}\} \cup \text{Exceptions}$

Enrole(p,er,o) \triangleq $\wedge \exists r_1 \in \text{Roles} : r_1 = \text{er}$
 $\wedge \forall r_2 \in \text{enroled} : r_2 \neq \text{er}$
 $\wedge \text{enroled}' = \text{enroled} \cup \{\text{er}\}$
 $\wedge \text{elements}' = \text{elements} \cup \{\langle p, \text{er}, o \rangle\}$
 $\wedge \text{UNCHANGED} \langle \text{guard}, \text{assert} \rangle$

Guard \triangleq $\wedge \forall r \in \text{Roles} : r \in \text{enroled}$
 $\wedge \text{guard}' = \text{GuardExpression}(\text{elements})$
 $\wedge \text{UNCHANGED} \langle \text{enroled}, \text{elements}, \text{assert} \rangle$

Begin \triangleq $\wedge \text{guard} = \text{TRUE}$
 $\wedge \forall r \in \text{Roles} : r.\text{state}' = \text{"started"}$
 $\wedge \text{UNCHANGED} \langle \text{enroled}, \text{elements}, \text{assert}, \text{guard} \rangle$

ExecuteRoles \triangleq $\wedge \forall r \in \text{Roles} : r.\text{state} = \text{"started"}$
 $\wedge \forall \langle p, r, o \rangle \in \text{elements} : r.\text{result}' = \text{ExecuteCommands}(\langle p, r, o \rangle)$
 $\wedge \text{UNCHANGED} \langle \text{enroled}, \text{assert}, \text{guard} \rangle$

Assertion \triangleq $\wedge \forall r \in \text{Roles} : r.\text{result} = \text{"ok"}$
 $\wedge \text{assert}' = \text{AssertionExpression}(\text{elements})$
 $\wedge \text{UNCHANGED} \langle \text{enroled}, \text{elements}, \text{guard} \rangle$

NormalEnd \triangleq $\wedge \text{assert} = \text{TRUE}$
 $\wedge \forall r \in \text{Roles} : r.\text{state} = \text{"ended"}$
 $\wedge \forall r \in \text{Roles} : r.\text{state}' = \text{"wait"}$
 $\wedge \text{enroled}' = \langle \rangle$
 $\wedge \text{elements}' = \langle \rangle$
 $\wedge \text{assert}' = \text{FALSE}$
 $\wedge \text{guard}' = \text{FALSE}$

Figure 3. TLA Specification of a DMI (part 1)

```

InterruptRoles  $\triangleq$   $\wedge \exists r_1 \in \text{Roles} : r_1.\text{result} \in \text{Exceptions}$ 
 $\wedge \forall r_2 \in \text{Roles} : \text{IF } r_2.\text{result} \notin \text{Exceptions}$ 
 $\quad \text{THEN } r_2.\text{result}' = \text{"interrupted"}$ 
 $\quad \text{ELSE } r_2.\text{result}' = r_2.\text{result}$ 
 $\wedge \text{UNCHANGED } \langle \text{enroled}, \text{assert}, \text{guard} \rangle$ 

ExceptionResolution  $\triangleq$   $\wedge \forall r \in \text{Roles} : r.\text{result} \neq \text{"ok"}$ 
 $\wedge \text{Resolve}(\text{enroled})$ 
 $\wedge \text{UNCHANGED } \langle \text{enroled}, \text{elements}, \text{assert}, \text{guard} \rangle$ 

ActivateHandler  $\triangleq$   $\wedge \forall r_1, r_2 \in \text{Roles} : (r_1.\text{result} = r_2.\text{result})$ 
 $\wedge \exists h \in \text{Handlers} : (\exists r \in \text{Roles} : r.\text{result} \in h.\text{Exc})$ 
 $\wedge \text{UNCHANGED } \langle \text{enroled}, \text{elements}, \text{assert}, \text{guard} \rangle$ 

ExceptionalEnd  $\triangleq$   $\wedge \forall r_1, r_2 \in \text{Roles} : r_1.\text{result} = r_2.\text{result}$ 
 $\wedge \neg \exists h \in \text{Handlers} : (\exists r \in \text{Roles} : r.\text{result} \in h.\text{Exc})$ 
 $\wedge \forall r \in \text{Roles} : r.\text{state} = \text{"ended"}$ 
 $\wedge \forall r \in \text{Roles} : r.\text{state}' = \text{"wait"}$ 
 $\wedge \text{enroled}' = \langle \rangle$ 
 $\wedge \text{elements}' = \langle \rangle$ 
 $\wedge \text{assert}' = \text{FALSE}$ 
 $\wedge \text{guard}' = \text{FALSE}$ 

Next  $\triangleq$   $\vee \exists p \in \text{Players} : (\exists er \in \text{Roles} : (\exists o \in \text{Objects} : \text{Enrole}(p, er, o)))$ 
 $\vee \text{Guard} \vee \text{Begin} \vee \text{ExecuteRoles} \vee \text{Assertion} \vee \text{NormalEnd} \vee$ 
 $\vee \text{ExceptionalEnd} \vee \text{InterruptRoles} \vee \text{ActivateHandler}$ 

Spec  $\triangleq$   $\text{Init} \wedge \square [\text{Next}] \langle \text{enroled}, \text{elements}, \text{assert}, \text{guard} \rangle$ 

```

THEOREM Spec \Rightarrow \square TypeInvariant

Figure 4. TLA Specification of a DMI (part 2)

When a role terminates by raising an exception, then all other roles have to be interrupted, causing the exception resolution algorithm to be enabled. The step that represents the interruption of roles is `InterruptRoles`. This step is enabled when at least one of the roles has raised an exception. The raising of an exception is represented in the value that the role's `result` assumes. If the value belongs to the set of `Exceptions`, then the `InterruptRoles` action is enabled. The step will then set the state of all roles, which did not raise an exception, to "interrupted". Even if a role has terminated it will be interrupted when another role raises an exception. The `InterruptRoles` actions is defined in TLA as:

$$\begin{aligned} \text{InterruptRoles} \triangleq & \wedge \exists r_1 \in \text{Roles}: r_1.\text{result} \in \text{Exceptions} \\ & \wedge \forall r_2 \in \text{Roles}: \text{IF } r_2.\text{result} \notin \text{Exceptions} \\ & \quad \text{THEN } r_2.\text{result}' = \text{"interrupted"} \\ & \quad \text{ELSE } r_2.\text{result}' = r_2.\text{result} \\ & \wedge \text{UNCHANGED } \langle \text{enroled}, \text{assert}, \text{guard} \rangle \end{aligned}$$

If exceptions were raised and there is no exception handler for the exception that resulted from the exception resolution algorithm, then the exceptional termination of a DMI is defined in the `ExceptionalEnd` action. This step is enabled when all roles have the same result and there is no exception handler for that result. This step changes the state of all roles to "wait", meaning that the roles are ready to be executed again. The sets `enroled` and `elements` are emptied. The TLA definition of `ExceptionalEnd` is:

$$\begin{aligned} \text{ExceptionalEnd} \triangleq & \wedge \forall r_1, r_2 \in \text{Roles}: r_1.\text{result} = r_2.\text{result} \\ & \wedge \neg \exists h \in \text{Handlers}: (\exists r \in \text{Roles}: r.\text{result} \in h.\text{Exc}) \\ & \wedge \forall r \in \text{Roles}: r.\text{state} = \text{"ended"} \\ & \wedge \forall r \in \text{Roles}: r.\text{state}' = \text{"wait"} \\ & \wedge \text{enroled}' = \langle \rangle \\ & \wedge \text{elements}' = \langle \rangle \\ & \wedge \text{assert}' = \text{FALSE} \\ & \wedge \text{guard}' = \text{FALSE} \end{aligned}$$

5. Related work

The semantics described in Section 4 deals with the basic rules of a DMI, i.e. pre and post-synchronisation, roles activation, exception handling, and roles interruption.

We did not attempt to describe formally the semantics of the execution of the role's commands. The way external objects guarantee ACID properties is also not described (a formal description of the ACID properties can be found in [21]). In [22], for example, formal description of properties for a mechanism similar to the DMI (Coordinated Atomic actions [11] - CA actions differ from DMIs in the way exceptions are handled during the interaction) is given in Temporal Logic. In [23], a formal approach is used to model and verify a safety-critical system (namely the fault-tolerant production cell) designed using CA actions. In order to model-checking, the state transition system corresponding to a CA action based design is expressed in SMV (Symbolic Model Checking) [24] and system properties expressed in CTL [25].

The COALA framework [26] is proposed to allow system developers to model systems using the CA action concept. Within this work a formalisation of the CA action concept is developed that uses CO-OPN/2:

an object-oriented language based on Petri nets and partial order-sorted algebraic specifications.

The ERT model (ERT stands for extraction, refusals and traces) is used for formalising the CA action concept [27]. Refusals and traces are terms coming from CSP; term extraction refers to a specific technique used to relate systems specified at different levels of abstractions.

A mathematical framework based on Timed CSP for representing the use of CA actions in real-time safety-critical systems is proposed in [28]. It allows the interactions between concurrently functioning equipment items to be modelled and their behaviour to be reasoned about in an abstract way. The framework models dynamic system structuring using CA actions by explicitly modeling synchronisation between items and the controlling system. Although the framework is not developed for dealing with erroneously behaving action participants, it allows for better understanding of the CA action concept and can be used in developing general models incorporating mechanisms supporting system safety.

6. Conclusion

The strategy of dealing with concurrent exceptions by enclosing them in a language mechanism presented in this paper, has been successfully applied to several case studies [29] [30] [31]. This paper has showed how TLA can be used to formally describe the semantics of a mechanism that implements this strategy. The formal model described in this paper is now being applied to a case study, which will be model-checked using TLA tools.

We believe that based on the description of the formal semantics presented in this paper, the process of implementing languages like Dependable Interacting Processes [5] [20], which include DMIs as a basic language construct will be greatly facilitated.

Acknowledgments

We would like to thank our colleagues from the Department of Computing Science at the University of Newcastle, Robert Stroud and Ian Welch, and from the University of Durham, Jie Xu, for several discussions that helped in formulating the dependable multiparty interaction concept. This work is supported by FAPERGS and CNPq/Brazil (grant number 520503/00-7). Alexander Romanovsky is supported by European IST DSoS project (IST-1999-11585). We also thank the reviewers for their contribution in making this a better paper.

References

- [1] Digital Equipment Corporation, Massachusetts, USA. *VAXELN Pascal language reference manual: Programming*, 1986.
- [2] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, 12(8):811–826, 1986.
- [3] V. Issarny. An exception handling mechanism for parallel object-oriented programming: Toward reusable, robust distributed software. *Journal of Object Oriented Programming*, 6(6):29–39, 1993.
- [4] International Standard for Organization. *Ada 95 Reference Manual - ISO/8652-1995*. ISO, 1995.
- [5] A. F. Zorzo. *Multiparty Interactions in Dependable Distributed Systems*. PhD thesis, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, 1999.
- [6] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
- [7] Y.-J. Joung and S. A. Smolka. A comprehensive study of the complexity of multiparty interaction. *Journal of ACM*, 43(1):75–115, 1996.
- [8] I. Forman and F. Nissen. *Interacting Processes - A multiparty approach to coordinated distributed programming*. ACM Publishers, 1996.
- [9] H.-M. Järvinen and R. Kurki-Suonio. Disco specification language: Marriage of actions and objects. In *11th International Conference on Distributed Computing Systems*, pages 142–151. IEEE CS Press, 1991.
- [10] N. G. Levenson. *Safeware: System, safety and computers*. Addison Wesley, Reading, MA, USA, 1995.
- [11] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *25th International Symposium on Fault-Tolerant Computing*, pages 450–457. IEEE Computer Society Press, 1995.
- [12] B. Randell. Systems structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

- [13] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 2nd edition, 1993.
- [14] M. Evangelist, N. Francez, and S. Katz. Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering*, 15(11):1417–1426, 1989.
- [15] C. T. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.
- [16] R. D. Tennent. *Semantics of Programming Languages*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991.
- [17] M. Hennessy. *The Semantics of Programming Languages: An elementary introduction using Structural Operational Semantics*. John Wiley & Sons, Chichester, UK, 1990.
- [18] L. Lamport. Specifying concurrent systems with TLA⁺. In M Broy and R. Steinbruggen, editors, *Calculational System Design*. IOS Press, Amsterdam, 1999.
- [19] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE CS Press, 1977.
- [20] A. F. Zorzo. A language construct for DMIs. In *II Workshop of Tests and Fault Tolerance*, Curitiba, PR, Brazil, 2000.
- [21] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.
- [22] 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. In *De Va - Design for Validation*, 2nd year, pages 3–15. ESPRIT Long Term Project 20072, 1997.
- [23] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, E. Canver A. F. Zorzo, and F. von Henke. Rigorous development of a safety-critical system based on coordinated atomic actions. In *29th International Symposium on Fault-Tolerant Computing*, pages 68–75. IEEE CS Press, 1999.
- [24] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.

- [25] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 16, pages 995–1072. Elsevier Science Publishers, 1990.
- [26] J. Vachon. *COALA: a design language for reliable distributed systems*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2000.
- [27] M. Koutny and G. Pappalardo. The ERT model of fault-tolerant computing and its application to formalisation of coordinated atomic actions. Technical Report 636, Department of Computing Science, Newcastle upon Tyne, UK, 1998. <http://www.cs.ncl.ac.uk/research/trs>.
- [28] S. Veloudis and N. Nissanke. *Modelling coordinated atomic actions in timed CSP*, volume 1926 of *Lectures Notes in Computing Science*, pages 228–239. Springer Verlag, Berlin, Germany, 2000.
- [29] A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In *14th ACM Conference on Object-Oriented Programming Systems, Languages and Applications - OOPSLA '99*, pages 435–446, Denver, CO, USA, 1999. ACM Press.
- [30] A. F. Zorzo, A. Romanovsky, B. Randell J. Xu, R. J. Stroud, and I. S. Welch. Using coordinated atomic actions to design safety-critical systems: A production cell case study. *Software: Practice and Experience*, 29(8):677–697, 1999.
- [31] A. Romanovsky and A. F. Zorzo. Coordinated atomic actions as a technique for implementing distributed GAMMA computation. *Journal of Systems Architecture - Special Issue on New Trends in Programming*, 45(9):79–95, 1999.