FACULDADE DE INFORMÁTICA
PUCRS - Brazil
http://www.inf.pucrs.br

Towards a Fomal Specification of Dependable Multiparty Interctions

A. F. Zorzo and B. Randell

TECHNICAL REPORT SERIES

Number 001
May, 2000

Contact:

zorzo@inf.pucrs.br

http://www.inf.pucrs.br/~zorzo

A. F. Zorzo is a senior lecturer at PUCRS/Brazil, where he started working in 1991. His main research topics are fault tolerance, distributed systems, and object-oriented systems. He is member of the Parallel and Distributed Group at the PUCRS. He is also the coordinator of the Research Centre for High Performance.

B. Randell has been a Professor of Computing science in the Department of Computing Science at the University of Newcastle upon Tyne, UK since 1969..

Published by the Campus Global – FACIN – PUCRS

Av. Ipiranga, 6681

90619-900 Porto Alegre – RS – Brazil

Towards a Formal Specification of Dependable Multiparty Interactions

A. F. Zorzo
Faculdade de Informática
Pontifícia Universidade Católica do RS
90619-900 Porto Alegre - RS - Brazil
zorzo@inf.pucrs.br

B. Randell
Department of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne - UK - NE1 7RU
brian.randell@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 had considered how failures in one or more participants of the multiparty interaction could be dealt with. In this paper, we present the *formal semantics* of dependable multiparty interactions. In particular, dependable multiparty interactions extend the notion of multiparty interaction to include facilities for handling concurrent exceptions that may be raised during the interaction.

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

1 Introduction

Parallel programs are usually composed of diverse concurrent activities, and communication and synchronisation patterns between these activities are complex and not easily predictable. Thus, parallel programming is widely regarded as difficult: [1], for example, says that parallel programming is "more difficult than sequential programming and perhaps more difficult than it needs to be". In addition to the normal programming concerns, the programmer has to deal with the added complexity brought about by multiple threads of controls: managing their creation and destruction and controlling their interactions via synchronisation and communication.

Furthermore, with the proliferation of distributed systems, computer communication activities are becoming more and more distributed. Such distribution can include pro-

cessing, control, data, network management, and security [2]. Although distribution can improve the reliability of a system by replicating components, sometimes an increase in distribution can introduce some undesirable faults. To reduce the risks of introducing faults when distributing applications, and of coping with residual faults, it is important that this distribution is implemented in an organised way.

As in sequential programming, complexity in distributed, in particular parallel, program development can be managed *by providing appropriate programming language constructs.* Language constructs can help both by supporting encapsulation so as to prevent unwanted interactions between program components and by providing higher-level abstractions that reduce programmer effort by allowing compilers to handle mundane, error-prone aspects of parallel program implementation [1].

A mechanism that encloses multiple processes executing a set of activities together is called a *multiparty interaction* [3] [4] [5] [6]. In a multiparty interaction, several executing processes somehow "come together" to produce an intermediate and temporary combined state, use this state to execute some joint activity, and then leave the interaction and continue their normal execution.

There has been a lot of work in the past years on multiparty interaction, but most of it has been concerned with synchronisation, or handshaking, between parties rather than the enclosure of several programmed activities executed in parallel by the interaction participants. For example, specification languages like CSP [7], LOTOS [8], or programming languages like Ada95 [9], only deal with synchronisation between processes. However, the programmer designing a set of processes that are taking part in a cooperating activity is left with full responsibility for ensuring that it is just these processes that are involved in the activity, and that they do not interfere with, or suffer interference from, other processes that are not supposed to be involved.

Interference from processes that are not involved in an interaction can lead to unexpected results, hence failure to deliver the requested service. Furthermore, to keep track of all processes that have interfered with the participants of an interaction is a very complex task. Usually, circumstances which may prevent an operation from providing its service are called *exceptions.* These exceptions have to be handled with care, since the state of a system can be inconsistent when their occurrence is detected [10]. The treatment of exceptions has been studied for many years and mechanisms for handling exceptions in sequential processes have been included in several programming languages [11] [9] [12] [13] [14] [15]. Although there has been a lot of research in treating exceptions in sequential pro-

cesses, language mechanisms for handling concurrent exception in programming languages are as yet in their early stages.

This paper shows a mechanism that integrates concurrent exception handling to the multiparty interaction concept. The paper also shows how this extended multiparty interaction concept, i.e. multiparty interaction with concurrent exception handling, can be embedded in programming languages. The extended multiparty interaction concept will be called *dependable multiparty interaction* and will be able to cope with several concurrent exceptions being raised during the multiparty interaction. Furthermore, formal semantics of this new concept is fully described using Temporal Logic of Actions (TLA) [16].

Section 2 describes the dependable multiparty interaction concept. Section 3 presents the Dependable Interaction Processes (DIP) language, which contains DMI as a basic language construc. Section 4 presents the formal description of DMIs.

2 Dependable Multiparty Interactions

As mentioned in Section 1, existing multiparty interaction mechanisms do not provide features for dealing with possible faults that may happen during the execution of the interaction. In some, the underlying system that is executing those multiparty interactions will simply stop the system in response to a fault. In DisCo [17], for instance, if an assertion inside an action is false, then the run-time system is assumed to stop the whole application. This situation is unacceptable in many situations.

In this section, both mechanisms are brought together, i.e. exception handling is added to multiparty interactions. This new mechanism is called a *dependable multiparty interaction* (DMI). Specifically, a DMI is a multiparty interaction mechanism that provides facilities for:

- HANDLING CONCURRENT EXCEPTIONS: when an exception occurs in one of the bodies of a participant, and not dealt with by that participant, the exception must be propagated to all participants of the interaction [18] [19]. 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 [20] [21]: 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: a participant 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.

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 the exception handling multiparty interactions are represented as one entity, a composite multiparty interaction since they are isolated from the outside in order that, for example, the raising of an exception is not seen by the enclosing context of a DMI.

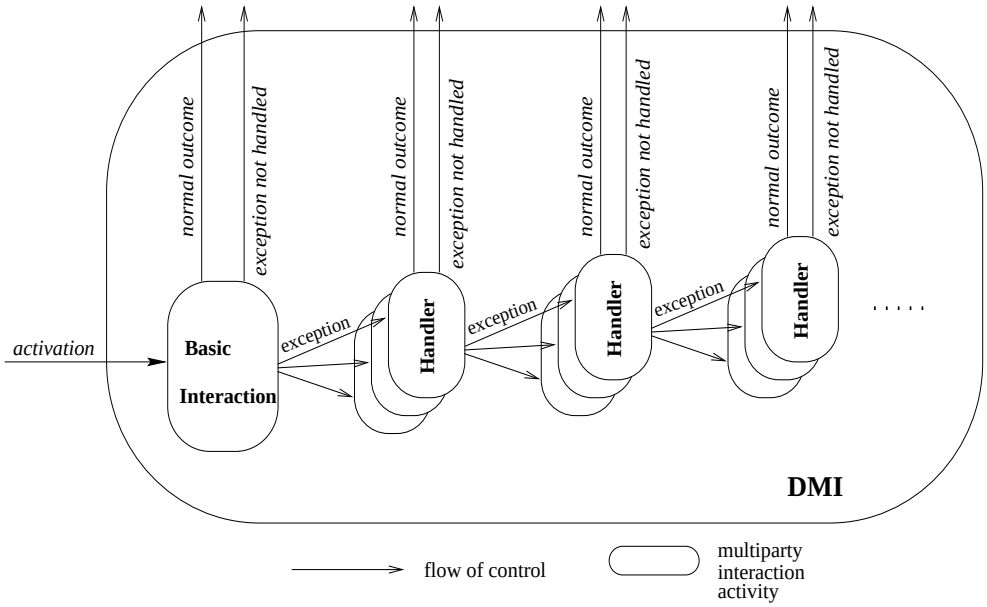

Figure 1: Dependable Multiparty Interaction

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 a exception resolution algorithm based on [18] 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 [3] and [6], we have made the following design choices for DMIs:

i) 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.

ii) The processes should synchronise their entry to and exit from the DMI.

iii) 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.

iv) The way the underlying system executes a DMI can be synchronous or asynchronous.

The choice for allowing a varying set of processes to enrole into a DMI is related to the expressive power of the language construct we intend to provide. In [3] 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. [22] 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 [6] 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 [6] in relation to the visibility of shared data inside the DMI. In our proposal, participants can exchange data inside the DMI, while in [6] participants of a multiparty interaction view shared data as "frozen" when the multiparty interaction commences.

The next section shows how the DMI concept can be embedded in a programming language. We based our description on Interacting Processes (IP) [4] and Distributed Cooperation (DisCo) [17]. This new language is a test bed for implementing DMIs. It provides DMIs as a basic language construct.

3 Dependable Interacting Processes - DIP

Dependable Interacting Processes (DIP) [23] is a language that allows a designer to specify exception handling in multiparty interactions. DIP extends languages like DisCo and IP (Interacting Processes), where exceptions are not considered in the specification of a system.

A program in DIP is composed of a set of objects O (instances of classes), a set of teams T (instances of actions), and a set of players P (instances of processes).

- $program = \{O,\ T,\ P\}$

Each program in DIP has a global state that is represented by the set of objects O. The objects that represent the global state of a DIP program are called *global objects*. Changes

to the global objects of a DIP program can only be made inside a team belonging to the set of teams T. Players in DIP are responsible for specifying the order in which the teams in T are executed, hence the order in which the state of a DIP program changes.

3.1 DIP Teams

Teams are used to describe dependable multiparty interactions in DIP. A team t in DIP is composed of a name, a main body b (called simply the body from now on) and a set of handler teams H. The body of a team and the handler teams are associated via the exceptions that can be raised inside the body of the team.

3.1.1 Team Body

Each body b is composed of: *i*) a set of roles R; *ii*) a set of objects O_{roles} that are manipulated by the roles, i.e. objects that are sent to the team as role parameters; *iii*) a set of local objects O_{local} that have the same semantics as global objects with respect to the roles in R, i.e. roles can modify these objects only inside another team; *iv*) a set of local teams LT (nested teams) used to modify the local objects in O_{local}[1]; *v*) a boolean expression, called *guard*, that checks the preconditions of the team and must be true in order for the roles of the team to start; *vi*) a boolean expression, called *assertion*, that checks the post-conditions of the team and must be true in order for the roles to finish normally; and a set of outcomes OUT it can produce, i.e. a normal outcome or one exception which is signalled to the callers of the team. The exceptions a team can signal are expressed as a list after the word **exceptions**. The list is structured as a tree, e.g. $e(e_1,e_2)$ represents a tree in which e is the parent of e_1 and e_2. Local objects and local (nested) actions are created when all roles become active and destroyed when the roles become inactive again.

The syntax of a team body is based on the syntax of IP teams, while the semantics of the use of objects is similar to DisCo actions. Teams in DIP differ from the IP teams in the sense that IP allows the static definition of processes that belong to that team, while in DIP the only static computation code allowed is the one inside the roles of the DIP team. The semantics of objects that are sent to a DIP team are very similar to the way objects are treated in DisCo actions, i.e. they can only be used in one team at a time.

[1] Objects in O_{roles} can also be modified by a team in LT

3.1.2 Team Guard and Assertion

Guard is a boolean expression, a precondition, over the objects that are carried to the team by the roles (O_{roles}). This boolean expression is tested only when all roles become active in an execution of a team. The *guard* states a necessary condition, not sufficient, for the team to start. If the *guard* does not hold, then an exceptional outcome is produced and it can be treated by a handler. The *guard* can be empty, having the same effect as if it is always true.

Assertion is a boolean expression, a post-condition, over the team's objects (O_{team}). This expression must be true in order for the team to finish normally. Similarly to the *guard*, if the *assertion* does not hold, then an exceptional outcome is produced and it can be treated by a handler. The *assertion* can be empty, having the same effect as if it is always true.

Team guard and assertion have similar meaning to guard and assertions in DisCo. There are some major differences though. First, an action in DisCo is only activated if the guard is true, while in DIP the *body of the team* is only activated if the guard is true. If the guard is not true in DIP, then an exception can be raised and a handler for that exception will be tried. Second, assertions in DisCo can be inserted anywhere inside an action, while in DIP, the assertion is used only for testing post-conditions, hence it is tested at the end of DIP team's execution. The third difference is that a DIP assertion can raise an exception if it is false, and an exception handler may be executed. In DisCo, if an assertion is false, then the system is assumed to stop.

3.1.3 Team Outcomes

A team can produce two different kinds of outcomes (results): *i*) normal, when all roles are activated, *guard* and *assertion* are satisfied; *ii*) exceptional outcome, when *guard* or *assertion* are false; when a role fails to perform its activity; or when an object being manipulated by a role has at least one of its assertions signalling an exception.

3.1.4 Team Roles

Roles are the means for describing computation inside a team. Each role r_i has a name, a set of objects, O_{role_i}, and a set of commands C_{role_i}. The objects used by the role are a subset of the objects of the team. Roles are passive entities but become **active** when players, or the roles in a containing team, activate them.

Roles in DIP, have a similar syntax to roles in IP teams, but their semantics differ greatly. For example, in IP, roles do not have to start at the same time, they are more like methods in object-oriented languages. Synchronisation can be achieved, in IP, by means of its interaction basic construct.

3.1.5 Team Handlers

Each team t can have an associated set of handlers h_i. Each handler is composed of a set of roles, a set of objects that are manipulated by the roles, a *guard*, an *assertion*, a set of exceptional outcomes, and a set of exceptions handled by this handler. A handler is activated when one of the exceptions it handles is raised in the body of a team or in another handler. Handlers can be used for several purposes: to recover a team from an error situation; to relax the guard of a team[2]; to relax the assertion of a team; to execute a new diverse version of a team with different guard, roles, and assertion; and so on. A handler has basically the same structure as the body of the team, but is activated by an exception or set of exceptions H_i. OUT_i is the set of outcomes the handler h_i can produce.

Connection. A team t is connected to a handler h_i if there is an exceptional outcome from t that is handled by h_i. We express a connection by $\rightarrow$. A handler h_i is connected to a handler h_j, $h_i \rightarrow h_j$, if an exceptional outcome of h_i is an exception handled by h_j. A team t is also connected to h_j if there is a sequence of connections that leads t to h_j, i.e. there is a sequence $t \rightarrow h_i \rightarrow ... \rightarrow h_k \rightarrow ... \rightarrow h_j$. If h_i is connected to h_j, then h_j cannot be connected to h_i, i.e. the connections produce a directed acyclic graph.

The set of roles R and the set of objects O_{roles} in a handler h_i are the same as in the team t that is connected to h_i. Each handler can be connected to several other handlers. *Guard* and *assertion* in a handler can be different from those in the team t. The set of exceptions that is handled by a handler is a subset of the exceptional outcome of the body of the team or handlers that are connected to it. If the commands of a handler role are the same as in the team t, then these commands are assumed in the handler role and do not need to be rewritten. The clause **same** after the declaration of a handler role specifies this property.

[2]E.g. imagine that a team needs two devices to execute an activity, but only one is available (the other may be broken), the guard will not be true, but a handler may have been provided that can nevertheless execute the same activity in a degraded mode.

The exceptional outcome of a team (OT) is composed of the union of the outcomes of the body of the team b and the outcomes of all handlers that b is connected to.

Concurrent exceptions raised in different roles are dealt with using the model proposed by [18]. When defining the exceptions the body of the team or one of its handlers can signal, the designer has to provide a hierarchy of exceptions in the format of a tree, so in the case of more than one exception being raised at the same time, the highest of the two will be handled, if there is a handler to treat that exception. In the case of the exceptions being on the same level of hierarchy, an immediately higher exception than the two will be handled. Two handlers cannot handle the same exception. If there is no handler for the raised exception, then it is signalled to the invokers of the team.

Actions

Teams are instances of actions. An action is a template description which specifies properties and behaviour for a set of similar teams. Actions are the way of defining the structure of a team presented in the previous section.

4 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. For example, in *operational semantics* the behaviour of an abstract processor is used to describe the effects of each language construct; in *axiomatic semantics* rules are provided to show the relevant state changes immediately after the execution of a language construct; and *denotational semantics* programs are defined in terms of mathematical functions which are used to derive properties of the original program. Several authors report how to use these approaches for describing the semantics of programming languages [24] [25] [26].

Concurrent systems are usually described in terms of their behaviour - what they do in the course of an execution [27]. The Temporal Logic formal model [28] 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 [16]. This variation is called Temporal Logic of Actions - TLA. TLA provides the mathematical basis for describing properties of concurrent systems.

In Section 4.1, an introduction to TLA is presented. In Section 4.1.1, the semantics of DMI is presented using TLA.

4.1 Temporal Logic of Actions

The Temporal Logic of Actions - TLA [16] 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. Semantically, an action is true or false for a pair of states, with primed variables referring to the second 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 $< s, t >$ satisfies that action.

Rather than presenting the full description of TLA, a simple program [16] in DIP 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.

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

Figure 2: Simple Program in DIP

The TLA formula for the above DIP process is defined as follows[3]:

$$\begin{array}{lll} \prod & \triangleq & \wedge \ \texttt{(x = 0)} \\ & & \wedge \ \Box\,\texttt{[x'} = \texttt{x + 1]}_{\texttt{x}} \\ & & \wedge \ \texttt{WF}_{\texttt{x}}\texttt{(x'} = \texttt{x + 1)} \end{array}$$

A TLA formula is true or false on a behaviour. Formula $\prod$, presented above, is true on a behaviour in which the i^{th} state assigns the value $i-1$ to `x`, for $i = 1, 2, ...$ In the above TLA formula, the conjunct `(x = 0)` specifies that initially, `x` is equal to `0`; the conjunct $\Box$`[x' = x + 1]`$_{\texttt{x}}$ specifies that the value of `x` in the next state (`x'`) is always ($\Box$) 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 `WF`$_{\texttt{x}}$`(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 practice, the specification of `WF` above implies that any step changes `x`.

Because TLA formulae can be as long as several pages (see [29]), TLA$^+$ was introduced to help in writing such large formulae. TLA$^+$ is a language for describing TLA formulae. Operators in TLA$^+$ are classified as constant operators, action operators, and temporal operators. (For further reading about TLA see [16] [27].)

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

4.1.1 DMI in TLA$^+$

In this section we will present the semantics of a dependable multiparty interaction. A DIP **action** is the command structure that represents a DMI.

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. The players send objects to the roles;
- a player has to activate a role in DMI in order to execute the commands inside a role;

[3]The symbol $\triangleq$ means *equals by definition.*

- 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 can be raised during the execution of a DMI. If exceptions are raised, then all roles that have not raised an exception are interrupted, and an exception resolution algorithm is executed when all roles have either 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 and predicates:

- `Exceptions`: the set of exceptions handled by the DMI;
- `Commands`: a set of commands;
- `Objects`: a set of objects;
- `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 <`p,er,o`>, 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`.

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:

```
Init ≜  ∧ ∀r ∈ Roles : r.state = "wait"
        ∧ guard = FALSE
        ∧ assert = FALSE
        ∧ enroled = {}
        ∧ elements = {}
```

The type invariant specifies that the `guard` and `assert` are `BOOLEAN` variables, the state of a role can 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:

```
TypeInvariant ≜  ∧ guard, assert ∈ BOOLEAN
        ∧ ∀r ∈ Roles : r.state ∈ {"wait","ended","started"}
        ∧ ∀r ∈ Roles : r.result ∈ {"ok","interrupted"} ∪ Exceptions
```

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 two first 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:

$$
\begin{array}{l}
\texttt{Enrole(p,er,o)} \triangleq \ \land\ \exists \texttt{r}_1 \in \texttt{Roles} : \texttt{r}_1 = \texttt{er} \\
\qquad\qquad\qquad\ \land\ \forall \texttt{r}_2 \in \texttt{Enroled} : \texttt{r}_2 \neq \texttt{er} \\
\qquad\qquad\qquad\ \land\ \texttt{enroled}' = \texttt{enroled} \cup \{\texttt{er}\} \\
\qquad\qquad\qquad\ \land\ \texttt{elements}' = \texttt{elements} \cup \{\texttt{<p,er,o>}\} \\
\qquad\qquad\qquad\ \land\ \texttt{UNCHANGED}\ \langle \texttt{guard, assert} \rangle
\end{array}
$$

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:

$$
\begin{array}{l}
\texttt{Guard} \triangleq \ \land\ \forall \texttt{r} \in \texttt{Roles} : \texttt{r} \in \texttt{enroled} \\
\qquad\qquad\ \land\ \texttt{guard}' = \texttt{GuardExpression(elements)} \\
\qquad\qquad\ \land\ \texttt{UNCHANGED}\ \langle \texttt{enroled, elements, assert} \rangle \\
\texttt{Begin} \triangleq \ \land\ \texttt{guard} = \texttt{TRUE} \\
\qquad\qquad\ \land\ \forall \texttt{r} \in \texttt{Roles} : \texttt{r.state}' = \texttt{"started"} \\
\qquad\qquad\ \land\ \texttt{UNCHANGED}\ \langle \texttt{enroled, elements, assert, guard} \rangle
\end{array}
$$

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:

$$
\begin{array}{l}
\texttt{ExecuteRoles} \triangleq \ \land\ \forall \texttt{r} \in \texttt{Roles} : \texttt{r.state} = \texttt{"started"} \\
\qquad\qquad\ \land\ \forall \texttt{<p,r,o>} \in \texttt{elements} : \texttt{r.result}' = \texttt{ExecuteCommands(<p,r,o>)} \\
\qquad\qquad\ \land\ \texttt{UNCHANGED}\ \langle \texttt{enroled, elements, assert, guard} \rangle
\end{array}
$$

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:

```
Assertion ≜ ∧ ∀r ∈ Roles : r.result = "ok"
            ∧ assert′ = AssertionExpression(elements)
            ∧ UNCHANGED ⟨enroled, elements, guard⟩
```

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:

```
NormalEnd ≜ ∧ assert = TRUE
            ∧ ∀r ∈ Roles : r.state′ = "wait"
            ∧ enroled′  = ⟨ ⟩
            ∧ elements′ = ⟨ ⟩
            ∧ assert′ = FALSE
            ∧ guard′ = FALSE
```

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:

```
ActivateHandler ≜ ∧ ∀r1,r2 ∈ Roles : (r1.result = r2.result)
                  ∧ ∃h ∈ Handlers : (∃r ∈ Roles : r.result ∈ h.Exc)
                  ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩
```

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:

```
MODULE DMI

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

Init ≜ ∧ ∀r ∈ Roles : r.state = "wait"
       ∧ guard = FALSE
       ∧ assert = FALSE
       ∧ enroled = {}
       ∧ elements = {}
TypeInvariant ≜ ∧ guard, assert ∈ BOOLEAN
                ∧ ∀r ∈ Roles : r.state ∈ {"wait","ended","started"}
                ∧ ∀r ∈ Roles : r.result ∈ {"ok","interrupted"} ∪ Exceptions
Enrole(p,er,o) ≜ ∧ ∃r_1 ∈ Roles : r_1 = er
                 ∧ ∀r_2 ∈ enroled : r_2 ≠ er
                 ∧ enroled' = enroled ∪ {er}
                 ∧ elements' = elements ∪ {<p,er,o>}
                 ∧ UNCHANGED ⟨guard, assert⟩
Guard ≜ ∧ ∀r ∈ Roles : r ∈ enroled
        ∧ guard' = GuardExpression(elements)
        ∧ UNCHANGED ⟨enroled, elements, assert⟩
Begin ≜ ∧ guard = TRUE
        ∧ ∀r ∈ Roles : r.state' = "started"
        ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩
ExecuteRoles ≜ ∧ ∀r ∈ Roles : r.state = "started"
               ∧ ∀<p,r,o> ∈ elements : r.result' = ExecuteCommands(<p,r,o>)
               ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩
Assertion ≜ ∧ ∀r ∈ Roles : r.result = "ok"
            ∧ assert' = AssertionExpression(elements)
            ∧ UNCHANGED ⟨enroled, elements, guard⟩
NormalEnd ≜ ∧ assert = TRUE
            ∧ ∀r ∈ Roles : r.state = "ended"
            ∧ ∀r ∈ Roles : r.state' = "wait"
            ∧ enroled' = ⟨ ⟩
            ∧ elements' = ⟨ ⟩
            ∧ assert' = FALSE
            ∧ guard' = FALSE
```

Figure 3: TLA$^+$ Specification of a DMI (part 1)

```
InterruptRoles ≜ ∧ ∃r1 ∈ Roles : r1.result ∈ Exceptions
                 ∧ ∀r2 ∈ Roles : IF r2.result ∉ Exceptions
                                 THEN r2.result′ = "interrupted"
                                 ELSE r2.result′ = r2.result
                 ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩

ExceptionResolution ≜ ∧ ∀r ∈ Roles : r.result ≠ "ok"
                      ∧ Resolve(enroled)
                      ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩

ActivateHandler ≜ ∧ ∀r1,r2 ∈ Roles : (r1.result = r2.result)
                  ∧ ∃h ∈ Handlers : (∃r ∈ Roles : r.result ∈ h.Exc)
                  ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩

ExceptionalEnd ≜ ∧ ∀r1,r2 ∈ Roles : r1.result = r2.result
                 ∧ ¬∃h ∈ Handlers : (∃r ∈ Roles : h.exception ≠ r.result)
                 ∧ ∀r ∈ Roles : r.state = "ended"
                 ∧ ∀r ∈ Roles : r.state′ = "wait"
                 ∧ enroled′ = ⟨ ⟩
                 ∧ elements′ = ⟨ ⟩
                 ∧ assert′ = FALSE
                 ∧ guard′ = FALSE

Next ≜ ∨ ∃p ∈ Players : (∃er ∈ Roles : (∃o ∈ Objects : Enrole(p,er,o)))
       ∨ Guard ∨ Begin ∨ ExecuteRoles ∨ Assertion ∨ End
       ∨ InterruptRoles ∨ ActivateHandler

Spec ≜ Init ∧ □[Next]⟨enroled, elements, assert, guard⟩

THEOREM Spec ⇒ □TypeInvariant
```

Figure 4: TLA$^+$ Specification of a DMI (part 2)

```
ExceptionResolution ≜ ∧ ∀r ∈ Roles : r.result ≠ "ok"
                      ∧ Resolve(enroled)
                      ∧ UNCHANGED ⟨enroled, elements, assert, guard⟩
```

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:

$$
\begin{array}{lll}
\texttt{InterruptRoles} \triangleq & \land & \exists \texttt{r}_1 \in \texttt{Roles} : \texttt{r}_1\texttt{.result} \in \texttt{Exceptions} \\
& \land & \forall \texttt{r}_2 \in \texttt{Roles} : \texttt{IF } \texttt{r}_2\texttt{.result} \notin \texttt{Exceptions} \\
& & \qquad \texttt{THEN } \texttt{r}_2\texttt{.result}' = \texttt{"interrupted"} \\
& & \qquad \texttt{ELSE } \texttt{r}_2\texttt{.result}' = \texttt{r}_2\texttt{.result} \\
& \land & \texttt{UNCHANGED } \langle \texttt{enroled, elements, assert, guard} \rangle
\end{array}
$$

4.2 Discussion

The semantics described in Section 4.1.1 deal 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 [30]). In [31], for example, formal description of enclosure properties for a mechanism similar to the DMI (CA action [32] - CA actions differ from DMIs in the way exceptions are handled during the interaction) is given in Temporal Logic. Furthermore, in [33] and [34] formal verification of safety and liveness properties of two safety-critical systems designed and implemented using CA actions was performed. DMIs can be used to implement CA actions, as described in [35]. Therefore, the same strategy to formal verify a system implemented with CA actions can be used to one implemented with DMIs.

5 Conclusion

The strategy of enclosing interactions in dependable systems presented in this paper, has been successfuly applied to several case studies [36] [35] [37] [38]. This paper has showed the formal semantics of a mechanism that implements this strategy, i.e. dependable multiparty interactions.

We believe that based on the description of the formal semantics presented in this paper, languages like Dependable Interacting Processes, which include DMIs as a basic

language construct can be more easily implemented.

Acknowledgements

We would like to thank our colleagues from the Department of Computing Science at the University of Newcastle, Robert Stroud, Alexander Romanovsky and Ian Welch, and from the University of Durham, Jie Xu, for several discussions that led to the dependable multiparty interaction concept. We also thank several members of the ESPRIT Long Term Research Project 20072 on "Design for Validation" (DeVa). This work was supported by CNPq/Brazil under grant number 200531/95.6.

References

[1] I. Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems*, 18(4):454–476, 1996.

[2] P. G. Neumann. Distributed systems have distributed risks. *Communications of the ACM*, 39(11):130, 1996.

[3] Y.-J. Joung and S. A. Smolka. A comprehensive study of the complexity of multiparty interaction. *Journal of ACM*, 43(1):75–115, 1996.

[4] I. Forman and F. Nissen. *Interacting Processes - A multiparty approach to coordinated distributed programming.* ACM Publishers, 1996.

[5] P. C. Attie, N. Francez, and T. X. Austin. Fairness and hyperfairness in multiparty interactions. *Distributed Computing*, 6(4):245–254, 1993.

[6] M. Evangelist, N. Francez, and S. Katz. Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering*, 15(11):1417–1426, 1989.

[7] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[8] E. Brinksa. *On the Design of Extended LOTOS - A Specification Language for Open Distributed Systems.* PhD thesis, University of Twente, the Netherlands, 1988.

[9] International Standard for Organization, editor. *Ada 95 Reference Manual - ISO/8652-1995.* ISO, 1995.

[10] F. Cristian. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*, chapter 4, pages 68–97. BSP Professional Books, Oxford, UK, 1989.

[11] J. Gosling, J. Bill, and G. Steele. *The Java language specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[12] B. Meyer. *Eiffel: The Language.* Prentice Hall, Englewood Cliffs, NJ, USA, 1992.

[13] B. H. Liskov and A. Snyder. Exception handling in clu. *IEEE Transactions on Software Engineering*, pages 546–558, November 1979.

[14] A. Goldberg and D. Robson, editors. *Smalltalk-80: The Language and Its Implementation.* Addison Wesley, Reading, MA, USA, 1983.

[15] J. D. Ullman, editor. *Elements of ML programming.* Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[16] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[17] 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.

[18] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, 12(8):811–826, 1986.

[19] A. Romanovsky, J. Xu, and B. Randell. Exception handling and resolution in distributed object-oriented systems. In *16th IEEE International Conference on Distributed Computing Systems*, pages 545–552. IEEE Computer Society Press, 1996.

[20] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. Coordinated atomic actions: from concept to implementation. Technical Report 595, Department of Computing Science, Newcastle upon Tyne, UK, 1997. http://www.cs.ncl.ac.uk/research/trs.

[21] A. Romanovsky. Extending conventional languages by distributed/concurrent exception resolution. *Journal of Systems Architecture*, 46(1):79–95, 2000.

[22] C. T. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.

[23] A. F. Zorzo. *Multiparty Interactions in Dependable Distributed Systems*. PhD thesis, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, 1999.

[24] R. D. Tennent, editor. *Semantics of Programming Languages*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[25] M. Hennessy, editor. *The Semantics of Programming Languages: An elementary introduction using Structural Operational Semantics*. John Wiley & Sons, Chichester, UK, 1990.

[26] C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 1998.

[27] L. Lamport. Specifying concurrent systems with tla$^+$. In M Broy and R. Steinbruggen, editors, *Calculational System Design*. IOS Press, Amsterdam, 1999.

[28] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE CS Press, 1977.

[29] L. Lamport. The windows win32 threads api specification. Technical report, Digital - Systems Research Center, Palo Alto, CA, USA, 1996. http://www.research.digital.com/SRC/personal/lamport/tla.

[30] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[31] 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 *DeVa - Design for Validation*, 2rd year, pages 3–15. ESPRIT Long Term Project 20072, 1997.

[32] 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.

[33] E. Canver. Formal verification of the caa-design of the production cell. In *DeVa - Design for Validation*, 2nd year, pages 357–383. ESPRIT Long Term Project 20072, 1997.

[34] E. Canver, D. Schwier, A. Romanovsky, and J. Xu. Formal verification of the caa-designs: The fault-tolerant production cell. In *DeVa - Design for Validation*, 3rd year, pages 229–258. ESPRIT Long Term Project 20072, 1998.

[35] 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*, Denver, CO, USA, 1999. ACM Press.

[36] A. F. Zorzo. Dependable multiparty interactions: A case study. In *29th Conference on Technology of Object-Oriented Languages and Systems - TOOLS29-Europe*, Nancy, France, 1999. IEEE Computer Society Press.

[37] 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.

[38] 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.