A Language Construct for DMIs

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

Abstract

Dependable Multiparty Interacion (DMI) has recently been introduced as a mechanism that provides reliable interaction between participants. Specifically, a DMI is a multiparty interaction mechanism that provides facilities for handling concurrent exceptions and assuring consistency upon leaving the interaction. This paper describes how the DMI mechanism can be added to a programming language.
Keywords: DMI, multiparty interaction, 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 processing, 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].

One of such language constructs is an extension to the multiparty interaction mechanism [3] [4] [5] [6] called *Dependable Multiparty Interaction* (DMI) [7]. This paper describes how a DMI mechanism can be embedded in a programming language. Section 2 presents the DMI mechanism and Section 3 presents a language called DIP (Dependable Interacting Processes) in which the DMI mechanism is a basic language construct.

2 Dependable Multiparty Interactions

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, 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, exception handling is added to multiparty interactions in order to provide some form of dealing with faults that may happen during the execution of the interaction. 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 [8] [9]. 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.
- 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.

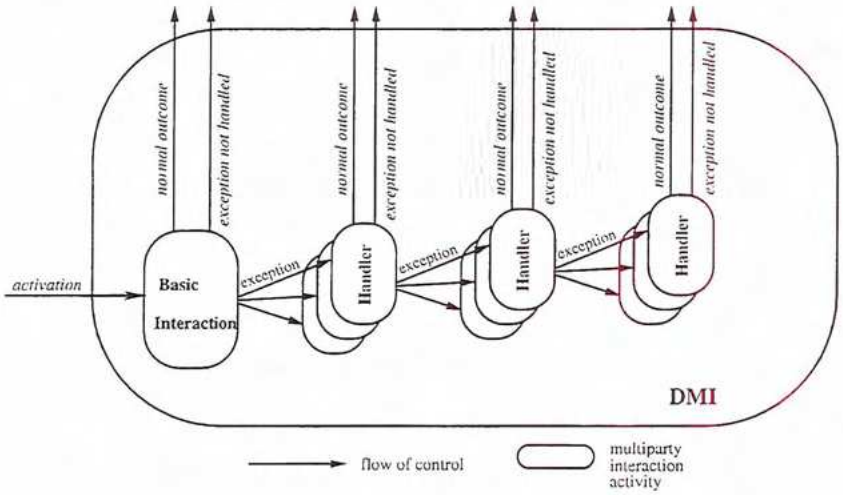

Figure 1: Dependable Multiparty Interaction

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 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.

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 [8] to decide which common exception will be raised and handled.

Further information about DMIs can be found in [7].

3 Dependable Interacting Processes - DIP

Dependable Interacting Processes (DIP) is a language that allows a designer to specify exception handling in multiparty interactions. DIP extends languages like DisCo [10] and IP [4] (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;* O_{global}. 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. Further information about DIP objects and DIP players can be found in [7].

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.

- $t = \{name_{team},\ b,\ H\}$
- $H = \{h_1, ..., h_n\},\ where\ n \geq 0$

An example of the team structure is in Figure 2.

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}; *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**, see Figure 2. 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.

- $b = \{R, O_{roles},\ O_{local},\ LT,\ guard,\ assertion,\ OUT\}$
- $O_{team} = O_{local} \cup O_{roles}$

```
action name_team is
    body is
        body b
    exceptions e(e1, e2)
    end body
    handler for e is
    body
        body of handler for exception e
    end body
    handler for e1 is
    body
        body of handler for exception e1
    end body
    handler for e2 is
    body
        body of handler for exception e2
    end body
end action
```

Figure 2: Team Structure in DIP

- $O_{roles} \subseteq O_{global}$

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.

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.

- *The body of a team* ***starts*** *when* <u>*all*</u> *roles are active and the guard of the body is true.*
- *If team* t_1 *and team* t_2 *are active, then* $O_{t_1} \cap O_{t_2} = \{\}$.

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.

- *The body of a team* ***terminates*** *normally when* <u>*none*</u> *of the roles of the body fail and the assertion of the body is true.*

In Figure 3, we show how a *guard* and an *assertion* can be inserted in the body of a team. The *guard-boolean-expression* and *assertion-boolean-expression* are expressions over the objects of the team, i.e. O_{team}

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,

```
action name_team is
    body is
     guard guard-boolean-expression
       ...
     assertion assertion-boolean-expression
    end body
end action
```

Figure 3: Guard and Assertion Declaration in DIP

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.

- $R = \{r_1, ..., r_m\}$, *where* $m \geq 1$
- $r_i = \{name_{role_i}, O_{role_i}, C_{role_i}\}$, *for* $1 \leq i \leq m$
- $O_{role_i} \subseteq O_{roles}$

An example of roles in a team is shown in Figure 4.

```
action name_team is
    body is
       role r1 with object1 is
          commands for role r1
       end role
       ..
       role rm with objectm is
          commands for role rm
       end role
    end body
end action
```

Figure 4: Roles Declaration in DIP

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[1]; 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.

- $h_i = \{b,\ H_i,\ OUT_i\}$

4 Conclusion

This paper has presented how a recently new concept for multiparty interactions that can handle concurrent exceptions, i.e. the *Dependable Multiparty Interaction* mechanism, could be embedded in a programming language. The language presented in this paper is called *Dependable Interacting Processes* (DIP). This new language is based on two existing languages: DisCo [10] and IP [4]. Tools for the language presented in this paper are currently under development.

Formal specification of the language construct presented in this paper can be found in [11].

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

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

[9] 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.

[10] 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.

[11] A. F. Zorzo and B. Randell. Towards a formal specification for dependable multiparty interactions. *Journal of Theoretical Computer Science*, page submitted, 2000.

[1] 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 more complex than the body of the team can execute the same activity in a degraded mode.