Åbo Akademi

OPEN ACCESS

This is an electronic reprint of the original article. This reprint may differ from the original in pagination and typographic detail.

Development of Fault Tolerant Grid Applications Using Distributed B

Boström, Pontus; Walden, Marina

Published: 01/01/2005

Document Version
Final published version

Link to publication

Please cite the original version:
Boström, P., & Walden, M. (2005). *Development of Fault Tolerant Grid Applications Using Distributed B*. (TUCS Technical Reports; Vol. 706). Turku Centre for Computer Science (TUCS). https://urn.fi/URN:NBN:fi-fe2022120168419

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

This document is downloaded from the Research Information Portal of ÅAU: 23. Apr. 2024

Pontus Boström | Marina Waldén

Development of Fault Tolerant Grid Applications Using Distributed B

Turku Centre *for* Computer Science

TUCS Technical Report
No 706, August 2005

Development of Fault Tolerant Grid Applications Using Distributed B

Pontus Boström
Åbo Akademi University, Department of Computer Science
Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14 A, 20520 Turku, Finland
Pontus.Bostrom@abo.fi

Marina Waldén
Åbo Akademi University, Department of Computer Science
Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14 A, 20520 Turku, Finland
Marina.Walden@abo.fi

TUCS Technical Report

No 706, August 2005

Abstract

Computational grids have become popular for constructing large scale distributed systems. Grid applications typically run in a very heterogeneous environment and fault tolerance is therefore very important for their correctness. Since the construction of correct distributed systems is difficult with traditional development methods we propose the use of formal methods. We use Event B as our formal framework, which we extend with new constructs such as remote procedures and notifications for reasoning about grid systems. The extended language, called Distributed B, ensures that the application can handle both node and network failures. Furthermore, the new constructs in Distributed B enable straightforward implementation of the specifications, as well as automatic generation of the needed proof obligations.

Keywords: Event B, Grid computing, Fault tolerance, Domain specific languages, Language extensions, Stepwise development

TUCS Laboratory
Distributed Systems Design Laboratory

1 Introduction

Computational grids have become a popular approach to handle vast amounts of available information and to manage computational resources. Examples of areas where grids have been successfully used for solving problems include biology, nuclear physics and engineering. Grid computing [8] is a distributed computing paradigm that differs from traditional distributed computing in that it is aimed toward large scale systems that even span organizational boundaries. Since grid applications run in a very heterogeneous computing environment fault tolerance is important in order to ensure their correct behaviour.

The development of correct grid applications is difficult with traditional software development methods. Hence, formal methods can be beneficial in order to ensure their correctness and structure their development from specification to implementation. The Action Systems formalism [4] is a formal method that is well suited for developing large distributed and concurrent systems, since it supports stepwise development. However, it lacks good tool support. The B Method [1], on the other hand, is a formal method provided with good tool support, but originally developed for construction of sequential programs. The B Method can be combined with Action Systems in order to formally reason about distributed systems as in the related methods B Action Systems [16] and Event B [2, 3]. B Action Systems models Action Systems in the B Method, while Event B also extends original B with new constructs. In this paper we use Event B as the basis for our work.

With generic formal languages like Event B, specifications are often unintentionally constructed in such a way that they cannot be implemented or are very difficult to implement efficiently. This can be due to synchronisation issues or the maintenance of atomicity of events. We have previously added extensions [5] to Event B to remedy this problem and to enable reasoning about grid applications using grid communication primitives. However, the extended Event B, referred to as Distributed B, did not consider fault tolerance. Here we will modify Distributed B to enable us to develop also fault tolerant grid applications.

The language Distributed B was designed for developing Grid applications using the Globus Toolkit [11] middleware. We here further develop this language to enable us to construct fault tolerant grid applications using only the fault tolerance mechanisms present in Globus toolkit. We add new constructs for handling exceptions raised during remote procedure calls and timeouts to handle lost notifications. These new features introduced to Event B force the developer to consider the grid environment and fault tolerance throughout the development. Furthermore, the constructs are introduced in such a manner that all needed proof obligations can be automatically generated and the system can be directly implemented.

In Section 2 we give an overview of the grid technology. Event B and the Distributed B extensions are presented in Section 3. In Section 4 we discuss the failure modes and the fault tolerance mechanisms used. Section

5 presents the extensions for developing fault tolerant grid applications. In Section 6 we present a case study using the new language. Implementation issues are discussed in Section 7 and in Section 8 we conclude.

2 Grid Systems

The purpose of grid systems is to share information and computing resources even over organizational boundaries. This requires security, scalability and protocols that are suited for Internet wide communication. The Open Grid Service Architecture (OGSA) [9] aims at providing a common standard to develop grid based applications. This standard defines what services a grid system should provide. A technical infrastructure specification defined by Open Grid Service Infrastructure (OGSI) [6] gives a precise definition of what a grid service is. The Globus Toolkit 3.x [11] is an implementation of the OGSI specification that has become de facto standard toolkit for implementing grid systems. We have chosen this toolkit as grid middleware for our extensions to Event B.

Grid systems usually have a client-server architecture, where the client initiates communication with the server that only responds to the client's request. A client may access several servers concurrently. In our grid applications a server is referred to as a grid service. Grid services as implemented in Globus Toolkit provide features such as remote procedures, notifications, services that contain state, transient services and service data. The main communication mechanism of grid services is remote procedure calls from client to grid service. If a call is unsuccessful the Globus toolkit will raise an exception for the programmer to handle. By using notifications a grid service can asynchronously notify clients about changes in its state. The state of grid services is preserved between calls and grid service instances can be dynamically created. Service data adds structured data to any grid service interface. This way not only remote procedures, but also variables are available to clients. Furthermore, Globus Toolkit contains an index service for managing information and keeping track of different types of services in the grid.

A grid application developed with Globus toolkit can be viewed as a collection of remote objects. A class defining a grid service can be used by first creating an instance of it with a specified grid service handle. These instances correspond to remote objects in Java RMI [14] or CORBA [14] and they can be used almost like normal local objects.

3 The Distributed B Extensions

Since constructing correct distributed applications is difficult with traditional development methods we here propose the use of formal methods to ensure their correctness. We have chosen to use Event B [2, 3], since it is a well supported formalism for modelling distributed systems. Event B is based

on the B Method by Abrial [1] and Action Systems by Back and Kurki-Suonio [4]. Grid applications can be specified in Event B. However, it is not straightforward to construct these specifications in such a manner that they can be efficiently implemented. Earlier we proposed extensions to Event B [5] for specifying and implementing grid applications. These extensions introduced grid features such as remote procedure calls and notifications to Event B. The semantics of the extensions is given by their translation to B. Note that we translate to B and not Event B, since the current tools for Event B also translate the specifications to B for verification.

We will here present the Distributed B extensions to Event B by a schematic example of a grid application. Furthermore, we show how these extensions are translated to B for verification. The abstract specification of an application is first written in Event B. Using a special grid refinement grid features are then introduced into the system in a stepwise manner. For a more formal treatment of the extensions the reader is referred to our previous paper [5].

3.1 Event B

The abstract specification of a grid application is given in a *system*-model written in Event B. A *system*-model contains constructs for defining sets, variables, an invariant, as well as events defining the behaviour of the component. The events consists of guarded substitutions. The semantics of the substitutions are given by the weakest precondition calculus introduced by Dijkstra [7]. When the guard of the event evaluates to *true* the event is said to be enabled. Enabled events are chosen non-deterministically for execution. When there are no enabled events left the event system terminates.

SYSTEM
  $\mathcal{C}$
VARIABLES
  v
INVARIANT
  $I(v)$
INITIALISATION
  $Init(v)$
EVENTS
$E_1 \hat{=}$
  ANY y **WHERE** $G_1(v, y)$ **THEN** $S_1(v, y)$ **END** ;
$E_2 \hat{=}$
  WHEN $G_2(v)$ **THEN** $S_2(v)$ **END**
END

The *variables*-clause defines a set of variables, v. The invariant $I(v)$ defines the type of these variables as well as additional properties that should be preserved during the execution of the system. The initialisation $Init(v)$ assign initial values to the variables. The *events*-clause contains the events, here E_1 and E_2, that describe the behaviour of the system. In event E_1 the values of the local variables y are non-deterministically chosen. When the predicate $G_1(v, y)$ evaluates to *true* the event is enabled and the substitution $S_1(v, y)$ can be executed. The event E_2 has no local variables and when $G_2(v)$

evaluates to *true*, $S_2(v)$ can be executed. When both guards $G_1(v, y)$ and $G_2(v)$ evaluates to *false* the event system terminates.

3.2 The Grid Service Machine

The *grid service machine* [5] has been introduced for giving the abstract specification of grid services. It extends the *system* model of Event B with constructs for specifying remote procedures and notifications to send. The grid service machine can be viewed as a class of which clients can obtain remote objects.

The example grid service machine presented below has a set of variables x, the remote procedure $Proc$ and the event E_A. Moreover, it can send notifications N_1 and N_2. A client can call the remote procedure and thereby enable the event. The event is then executed concurrently with the events of the client. When all the events (here only one) in the grid service has become disabled a notification is sent to the client.

In order to give our new constructs meaning and to use the tool support of B the grid service machine is translated to an ordinary B machine. Throughout this subsection the grid service machine is presented in the left column and its translation in the right one.

The B translation of the grid service machine generates an additional set, a constant and a variable. These are needed to model the instance management in B.

GRID_SERVICE	**MACHINE**
$\mathcal{A}$	$\mathcal{A}_V$
VARIABLES	**SETS**
x	A_INSTS
INVARIANT	**CONSTANTS**
$x_i \in T \wedge I_A(x)$	A_null
INITIALISATION	**PROPERTIES**
$Init_A(x)$	$A_null \in A_INSTS$
	VARIABLES
	x, A_Insts
	INVARIANT
	$A_Insts \subseteq A_INSTS \wedge$
	$A_null \notin A_Insts \wedge$
	$x_i \in A_Insts \rightarrow T \wedge$
	$I_{AV}(x, A_Insts)$
	INITIALISATION
	$x_i := \varnothing \parallel$
	$\ldots$
	$A_Insts := \varnothing$

The deferred set A_INSTS models all possible instances of grid service machine $\mathcal{A}$. The constant A_null models the empty instance, while the variable A_Insts models the set of instances that are currently in use by the client. Note that a grid service machine instance can only be accessed from one client. The variables are translated to take into consideration the instances. Each variable becomes a function from instance to the type defined in the grid service machine. The variables in the translation are initialised to empty sets. The instance will be initialised to $Init_A(x(inst))$ according

to the initialisation clause of the grid service machine in the constructor of the instance.

The remote procedures are translated to take the instance as an additional parameter. This is needed since a remote procedure call is always made to an instance of a grid service machine. Additionally, for each variable x_i in $\mathcal{A}$, a remote procedure, $GetX_i$, for enabling a client to read the variable value is automatically introduced. These procedures can then be used by a client as normal remote procedures. The events are then also translated to take into account every possible instance via an *any*-statement.

REMOTE_PROCEDURES
$Proc(p) \hat{=}$
  PRE $P(p)$
  THEN $S_p(x, p)$
  END ;

EVENTS
$E_A \hat{=}$
  WHEN $G_A(x)$
  THEN $S_A(x)$
  END ;

OPERATIONS
$Proc(inst, p) \hat{=}$
  PRE $inst \in A_Insts \wedge P(p)$
  THEN $S_p(x(inst), p)$
  END ;
$rr \leftarrow GetX_1(inst) \hat{=}$
  PRE $inst \in A_Insts$
  THEN $rr := x_i(inst)$
  END
...
$E_A \hat{=}$
  ANY $inst$ **WHERE**
   $inst \in A_Insts$
  THEN
    WHEN $G_A(x(inst))$
    THEN $S_A(x(inst))$
    END
  END

Notifications to be sent are defined in the *notifications*-clause.

NOTIFICATIONS
$N_1 \hat{=}$
  GUARANTEES $Q_1(x)$
  END ;
$N_2 \hat{=}$
  GUARANTEES $Q_2(x)$
  END ;
END

Each notification consists of a *guarantees*-statement. The *guarantees*-statement means that a certain condition $Q_i(x)$ holds in this grid service when the corresponding notification is sent. The notification handling is performed in the client. However, an invariant is added in the grid service machine to ensure that when the event system terminates the *guarantees*-statement of at least one notification evaluates to *true*, $\forall inst.(inst \in A_Insts \wedge \neg G_A(x(inst)) \Rightarrow (Q_1(x(inst)) \vee Q_2(x(inst))))$.

The translated B machine contains two additional automatically generated operations. These are the constructor and destructor of instances.

$$
\begin{array}{l}
r \leftarrow A_GetNew \mathrel{\hat{=}} \\
\quad \textbf{ANY } inst \textbf{ WHERE} \\
\quad\quad A_Insts \neq A_INSTS - \{A_null\} \wedge \\
\quad\quad inst \in A_INSTS - A_Insts \wedge \\
\quad\quad inst \neq A_null \\
\quad \textbf{THEN} \\
\quad\quad A_Insts := A_Insts \cup \{inst\} \parallel \\
\quad\quad Init_A(x(inst)) \parallel \\
\quad\quad r := inst \\
\quad \textbf{END} \ ; \\
\\
A_Destroy(inst) \mathrel{\hat{=}} \\
\quad \textbf{PRE } inst \in A_Insts \\
\quad \textbf{THEN} \\
\quad\quad A_Insts := A_Insts - \{inst\} \\
\quad\quad x_i := \{inst\} \lhd x_i \parallel \\
\quad\quad \ldots \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

The constructor adds the instance to the set of used instances and initialises it. In order to find instances to use, the addresses of all the instances are stored in the index service of Globus toolkit. The client can then locate a new instance by asking the index service to return the address to new free ones. The destructor of instances removes the instance from the set of used instances and modifies the variables x to reflect this change.

3.3 The Grid Refinement Machine

One of the benefits of Event B is that it supports refinement and thereby it enables stepwise development of systems. Refinement of a component preserves the behaviour of the abstract component while making it more concrete by adding variables and additional behaviour (events). An event is said to refine another event if the guard is strengthened and the behaviour of the substitutions is preserved.

In order to refine grid service machines and for introducing grid features to ordinary Event B specifications we introduce a *grid refinement machine*. The grid refinement machine extends the ordinary refinement of Event B with constructs for accessing grid service machine instances, refining remote procedures and handling notifications. The instances of grid service machines are modelled as ordinary variables. Remote procedure calls are modelled by ordinary operation calls. When all events in a grid service machine instance have become disabled a notification is received from it. The notification handlers in the grid refinement are then executed once for each notification. In order to verify that a grid refinement machine is a refinement of a more abstract specification it is translated to a refinement machine in B.

Grid refinement machines are here presented with the example $\mathcal{C}_1$ that uses instances of grid service machine $\mathcal{A}$ presented in Subsection 3.2. This grid refinement is a refinement of the abstract system $\mathcal{C}$ presented in Subsection 3.1. The grid refinement $\mathcal{C}_1$ contains the old variables v from $\mathcal{C}$, new variables w and a set of instances $a_1, \ldots, a_n$ of grid service machine $\mathcal{A}$. The

events E_1 and E_2 in the abstract specification are refined by more concrete events. The grid refinement machine also contains a new event F that only modifies the new variables w and a notification handler $N_1Handler$ for handling notifications of type N_1 from instances of $\mathcal{A}$. A notification handler is also considered to be a new event and, hence, it can only assign to new variables. The grid refinement machine $\mathcal{C}_1$ is given below in the left column and its translation to B in the right.

In Distributed B a new structuring mechanism [5], *references*, is used in the grid refinement to handle instances of grid service machines. This construct is translated into an *includes* statement in B. The concurrent execution of the events in the grid service machine instances and grid refinement is modelled by promoting all the events in the grid service machine into the refinement in the B model.

GRID_REFINEMENT	**REFINEMENT**
$\mathcal{C}_1$	$\mathcal{C}_{1V}$
REFINES	**REFINES**
$\mathcal{C}$	$\mathcal{C}$
REFERENCES	**INCLUDES**
$\mathcal{A}$	$\mathcal{A}_V$
VARIABLES	**PROMOTES**
$v, w, a_1, \ldots, a_n$	E_A
INVARIANT	**VARIABLES**
$a_1 \in \mathcal{A} \wedge$	$v, w, a_1, \ldots, a_n, A_notif$
$\ldots$	**INVARIANT**
$a_n \in \mathcal{A} \wedge$	$a_1 \in A_INSTS \wedge$
$J(v, w, a_1, \ldots, a_n)$	$a_1 \in A_Insts \cup \{A_null\} \wedge$
	$\ldots$
	$J_V(v, w, a_1, \ldots, a_n) \wedge$
	$A_notif \in A_Insts \rightarrow BOOL$

We introduce variables modelling instances of $\mathcal{A}$, a_i. The type of the variables a_i are given in the grid refinement as $\mathcal{A}$ which is then translated to the corresponding representation of instances in the traditional B model. The invariant J is also modified by the translation to take into account the changed representation of instances.

The notification handler can only be executed once for each notification from $\mathcal{A}$. Hence, we introduce variable A_notif of type function from the instances in use to boolean values. If the value is *true* for an instance, notifications can be received from it. If the value is *false* the notification has already been handled.

The initialisation of the grid refinement is similar to its B translation. The automatically generated variable A_notif is initialised to the empty set, which corresponds to the value of A_Insts.

INITIALISATION	**INITIALISATION**
$a_i := A_null \parallel$	$a_1 := A_null \parallel$
$\ldots$	$\ldots$
$a_n := A_null \parallel$	$Init'(v, w) \parallel$
$Init'(v, w)$	$A_notif := \varnothing$

Events of the grid refinement and the B translation are again similar.

```
EVENTS
E1 ≙
   ANY y WHERE
      G'1(v, y)
   THEN
      S'1(v, y);
      a_i ← A_GetNew
   END ;
E2 ≙
   WHEN G'2(v, w)
   THEN S'2(v, w)
   END ;

F ≙
   WHEN H(v, w, a_i)
   THEN
      S_n(w) ||
      a_i.Proc(f(v, w))
   END ;
```

```
OPERATIONS
E1 ≙
   ANY y WHERE
      G'1(v, y)
   THEN
      S'1(v, y);
      a_i ← A_GetNew;
      A_notif(a_i) := TRUE
   END ;
E2 ≙
   WHEN G'2(v, w)
   THEN S'2(v, w)
   END ;

F ≙
   WHEN H(v, w, a_i)
   THEN
      S_n(w) ||
      Proc(a_i, f(v, w));
      A_notif(a_i) := TRUE
   END ;
```

The remote procedure calls are translated in such a way that the instance is given as the first parameter to the procedure. Furthermore, the assignment $A_notif(inst) := TRUE$ is added after each call to a manually defined remote procedure and after a call to A_GetNew. This is done to enable the correct handling of notifications originating from that instance. However, it is not added after calls to the automatically generated remote procedures $GetX_i$, since they are read-only procedures that do not modify the state of the instance. Note that event above E_2 does perform any remote procedure calls and hence it does not contain assignment $A_notif(inst) := TRUE$.

Finally we study the notification handlers.

```
NOTIFICATION_HANDLERS
N1Handler ≙
   NOTIFICATION N1
   SOURCE inst ∈ 𝒜
   THEN T_N1(inst, w, a_i, a_j)
   END
END
```

```
N1Handler ≙
   ANY inst WHERE
      inst ∈ A_Insts∧
      A_notif(inst) = TRUE∧
      ¬G_A(x(inst)) ∧ Q1(x(inst))
   THEN
      T_N1(inst, w, a_i, a_j);
      A_notif(inst) := FALSE
   END
END
```

Notifications of type N_1 from instance *inst* of $\mathcal{A}$ are handled when all the events in the instance have become disabled, $\neg G_A(x(inst))$, and the condition in the *guarantees*-statement for that notification holds, $Q_1(x(inst))$. The variable A_notif is assigned *false* for this instance to denote that the notification has been handled.

3.3.1 Proof obligations

Since we develop grid applications in a stepwise manner, we need to show that each new specification is a refinement of the specification developed in

the previous step. In order to show that an Event B or a Distributed B component is a refinement of another component the following properties must hold [2, 16]:

1. The initialisation of the concrete specification has to be a refinement of the initialisation of the abstract specification.

2. All events in the abstract specification have to be refined by corresponding events in the concrete specification.

3. New events that refine *skip* can be introduced in the concrete specification.

4. The new events have to terminate when executed in isolation.

5. The concrete system is not allowed to terminate more often than the abstract system.

6. The guard of a remote procedure cannot be strengthened.

The proof obligations for rules 1-5 [2, 16] can be automatically generated by the tools of Event B. The proof obligation for rule 6 [16] needs some additional constructs. Alternatively, the developer can be restricted to only use non-guarded statements in the remote procedures. The proof obligations can then be discharged by the automatic and interactive provers for B.

4 Fault Tolerance using Globus Toolkit

Grid applications run in a very heterogeneous computing environment. This means that fault tolerance is highly important for the correct behaviour of the application. In this paper we develop extensions to Event B for developing fault tolerant grid applications using the fault tolerance mechanisms in Globus toolkit. Currently the only fault tolerance mechanism in Globus toolkit is exceptions due to failed remote procedure calls. More advanced fault tolerance mechanism such as Replication [14] and Check pointing [14] are not yet supported by the toolkit. Even with support for advanced fault tolerance mechanisms in the middleware the application needs to consider faults, since these mechanisms in the middleware might not be sufficient to handle the faults transparently.

It is not feasible to construct a system that can tolerate all types of faults [14]. We will limit the fault tolerance of Distributed B to handle two types of faults, which we consider to be most important: Firstly, a grid service instance can stop (crash) and be restarted. Secondly, network connections can fail to deliver messages to desired destinations. We do not consider situations where grid services does not satisfy its specification or where an attacker can deliberately force a grid service instance to produce erroneous results since we have proved our application correct and Globus toolkit has

a very comprehensive security infrastructure. Furthermore, the grid middleware can use TLS or other secure protocols for communication and therefore we can assume that data has not been corrupted during transmission.

4.1 Faults and Fault Detection

We can identify five distinct faults that can occur in remote procedure calls from a client to a server grid service instance [14]:

1. The server grid service instance has crashed before the call.
2. The network connection fails when calling a remote procedure.
3. The server instance fails during the call.
4. The network connection fails when returning the result.
5. The client crashes during the remote procedure call.

In the client the faults 1-4 are not easily separated from each other and they are handled by not using the failed server grid service instance anymore. These faults are directly detected in the client by the Globus toolkit middleware that then raises an exception. However, the server grid service instance can crash and be restarted and it needs to detect the crash in order to respond correctly to the client. A restart can be detected if the instance creates a file on startup using its address as the filename. If the file creation fails due to the fact that the file already exists, it can be assumed that the grid service has been restarted. This holds, since file creation is atomic if the correct method is used.

In order to handle the fifth fault above the called instance needs to be able to identify when the calling client has failed, i.e., the instance has become an *orphan* [14]. To discover this we introduce an *is-alive check* in the client and an *is-alive timeout* in the server grid service instance. We first introduce a new remote procedure in every grid service instance called *isAlive*. This remote procedure returns *true,* if the instance is accessible. In the client the *is-alive check* for a given grid service instance consists of calling remote procedure *isAlive* of the instance periodically with a specified time interval. If the call fails, an exception is raised by the Globus toolkit and the client knows that there is a problem with the instance or with the network connection. The client then stops using the instance. If the *isAlive* procedure was not called with the specified time interval in the server grid service instance, an *is-alive timeout* is triggered. This means that the client has either failed or stopped using the instance. The server grid service instance is then reset in order to enable other clients to use it. It is important to note that the times for the timeout mechanism should be chosen so that the client will get the exception for a failed call to *isAlive* before the server resets itself. This can be accomplished by choosing the timeout in the server grid service instance to be greater than the time period that the client calls

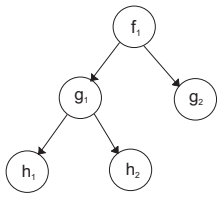

Figure 1: The tree formed by the grid service instances

isAlive with an appropriate amount of buffer time. This check for orphans is implemented in a layer on top of Globus toolkit that then provides an easy to use interface to Distributed B.

There is an additional fault, not directly related to remote procedure calls, that needs to be considered when allocating new grid service instances. There might not be any available grid service machine instances of the correct type when the client tries to obtain new instances to use. The problem can be due to broken network connections, a broken index service or exhaustion of the pool of available services. Note that the index service is here a single point of failure for the grid application. Hence, if it fails, the application might not function correctly. The index service could, however, be transformed into a replicated index service on several computers or we could use a peer-to-peer index service. The instance allocation is also implemented in a layer on top of Globus toolkit to present a more easy to use interface to Distributed B.

4.2 Use Cases Describing the Fault Tolerance Mechanisms

In order to clarify the behaviour of the system in the presence of faults we here present the use cases for the two most complicated failure modes. We consider a tree of grid service instances as shown in Figure 1. The grid service instance f_1 references instances g_1 and g_2, where g_1 then references h_1 and h_2.

4.2.1 The Middle Node Crashes

The first use case describes the actions taken when the grid service instance g_1 crashes. The instance g_1 is here a middle node that has client f_1 and has references to grid service instances h_1 and h_2.

1. Instance g_1 crashes. There are now two cases to consider, steps 2 and 3.

2. If the instance g_1 is unreachable from its client, instance f_1:

   (a) An exception is raised in f_1, due to a failed remote procedure call or *is-alive check*. (A notification timeout can also be triggered by this condition).

(b) The instance g_1 is removed from the set of used instances in f_1.

3. If the instance g_1 is reachable from f_1 but it has recovered from a failure and is in a restarted state:

   (a) The instance g_1 waits for an *is-alive timeout*. It raises exceptions for the clients remote procedure calls and *is-alive checks*. That way g_1 can make sure that the client f_1 knows that g_1 has failed.

   (b) Instance f_1 detects that instance g_1 has failed and it removes it from the set of used instances.

   (c) The instance g_1 receives an *is-alive timeout* and resets itself.

4. The instances h_i notice that g_1 has failed when they receive *is-alive timeouts* and they are then reset.

4.2.2 The Network Connection Fails.

The second use case describes the behaviour of the system when a network connection failure between, the client f_1 and the middle node g_1 is noticed.

1. The network connection between f_1 and g_1 fails.

2. An exception is raised in f_1 due to failed remote procedure call or *is-alive check*.

3. The instance f_1 removes g_1 from its set of used instances.

4. The instance g_1 receives an *is-alive timeout*, since its services are not requested anymore. It then resets itself.

5. The instances h_i reset themselves when they receive *is-alive timeouts* due to the reset of g_1.

4.2.3 Final Notes

What applies to a middle node g_i also applies to a leaf node h_i. The only difference is that we do not have to consider any child nodes. If the root node f_1 crashes the program terminates and no result is obtained.

5 Fault Tolerance in Distributed B

Previously we have developed a language called Distributed B [5] based on Event B for constructing fault free grid applications. In this section we modify the language to also incorporate fault tolerance mechanisms as described in the previous section, Section 4. Distributed B provided a way to implement grid applications and the fault tolerance mechanisms will also be introduced in a manner that ensures that they can be implemented.

5.1 Specification of Instance Management

A challenging problem when introducing fault tolerance is the management of instances. In order to correctly reason about instances we need to have a model of the instance management. The model in the original version of Distributed B is described in Subsection 3.2. During the translation of grid service machine $\mathcal{A}$ to original B we there added features such as a set A_INSTS modelling all instances of $\mathcal{A}$, a constant A_null modelling the empty instance, a variable A_Insts modelling the instances in use, as well as a constructor and destructor of instances. This model of the instance management is not changed when we introduce fault tolerance into Distributed B. However, we did not specify how an instance should be treated if a fault was encountered during a remote procedure call to it or if notification was never received from it. In order to also handle faults, an instance a_i of $\mathcal{A}$ is immediately removed from the set of instances in use, A_Insts, if a problem with it is encountered.

The model of instance management described above serves as an abstract specification that need to be implemented. To increase the confidence in the system it can be developed in a stepwise manner using, e.g., Event B. The model is refined to consider the behaviour of grid service instances, the communication between them, as well as the faults described in Section 4.

We here give an overview of a refined, more concrete model of the behaviour of an instance of $\mathcal{A}$ using the statechart diagram shown in figure 2. The users of fault tolerant Distributed B will not have to consider this refined model of instance management, since it is built into the language. It only serves as an illustration on how the instance management is implemented using Globus toolkit. The instance can be in three different states: *idle* when the instance is not reserved by a client; *busy* when a client has reserved the instance and *restarted* when the instance has detected that it has been restarted. The event *reserve* is generated when a client reserves the instance by a call to the operation *A_ GetNew*. The reservation is modelled by the transition from state *idle* to state *busy*. When the event *reserve* occurs the instance is initialised, modelled in Dsitributed B by executing the *initialisation*-clause of the grid service machine. The transition *release* from state *busy* to state *idle* models that the client releases the instance by a call to operation *A_ Destroy*. The event *ia_ timeout* denotes that an *is-alive timeout* occurred and the instance is reset, i.e., it enters the state *idle*. The *restart* event models restart of the grid service instance by taking the instance to state *restarted*. When a grid service instance belongs to the set of instances in use, A_Insts, it is in either state *busy* or *restarted,* since it has then been reserved by a client. Note that the instance will raise an exception when its remote procedures are called in state *restarted*.

5.2 Extensions to Distributed B

In order to handle faults as described in Section 4 and to handle instances as described in the previous subsection we add new constructs to the original

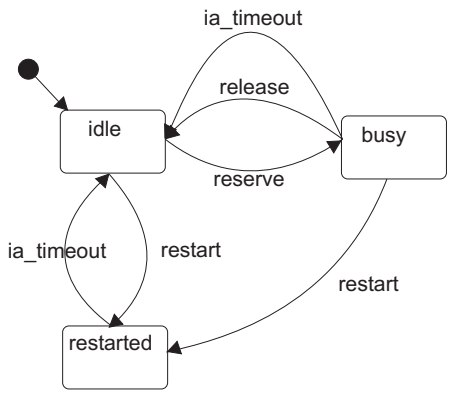

Figure 2: The behaviour of a grid service instance

version Distributed B. New constructs are needed for handling exceptions raised from remote procedures and timeouts for notifications. We also add constructs for handling orphans with the *is-alive check*. The mechanism for handling faults only affects the grid refinement machine and hence, the constructs in the grid service machine remains the same. For simplicity we will here consider only grid applications that do not have middle nodes. Hence, a refinement of a grid service machine cannot here reference other grid service machines.

As an example of a fault tolerant grid refinement machine we modify grid refinement $\mathcal{C}_1$ in Subsection 3.3 to handle faults. The fault tolerant grid refinement $\mathcal{C}_{FT}$ has the same variables (v, w) and instances ($a_1, \ldots, a_n$) as $\mathcal{C}_1$. The remote procedure calls to instance a_i are modified to consider faults. The notification handling mechanism is also modified to consider timeouts due to lost notifications. Furthermore, we need an additional event to handle exceptions raised by *is-alive checks*. The fault tolerant Distributed B models are again translated to B for verification purposes. Below the fault tolerant Distributed B constructs are given in the left column and their translation to B is given to the right. The complete example can be found in Appendix A.

5.2.1 Remote Procedures

Remote procedure calls can fail in several ways as outlined in Subsection 4.1. We add a *call*-substitution for calling remote procedure in order to model the exception handling mechanism present in Java and Globus toolkit.

```
call_subst::=''CALL'' operation_call
             ''EXCEPTION'' NG_Substitution
             ''END''
```

The *call*-part of this construct contains a remote procedure call. The *exception*-part then gives the non-guarded substitution that is executed if an

exception is raised during the call. The *call*-substitution is used whenever a remote procedure in a grid service instance is called as shown in event F. The *call*-substitution also needs to be used when calling A_GetNew to allocate a new instance since it can fail as described in Subsection 4.1.

$F \mathrel{\hat{=}}$
  WHEN $H(v, w, a_i)$
  THEN
    $S_n(w) \parallel$
    CALL $a_i.Proc(f(v, w))$
    EXCEPTION $T_f(w, a_i)$
    END
  END

$FOk \mathrel{\hat{=}}$
  WHEN $H(v, w, a_i)$
  THEN
    $S_n(w) \parallel$
    $Proc(a_i, f(v, w));$
    $A_notif(a_i) := TRUE$
  END ;

$FFail \mathrel{\hat{=}}$
  WHEN $H(v, w, a_i)$
  THEN
    $S_n(w) \parallel$
    $A_Destroy(a_i);$
    $A_notif := \{a_i\} \lhd A_notif;$
    $T_f(w, a_i)$
  END ;

The event F is translated to two separate operations in B, FOk and $FFail$, which both refine the abstract specification of F. One operation, FOk, models the successful execution of the remote procedure and the other, $FFail$, models the failed execution [2]. The operation FOk is translated as F previously in Distributed B, i.e., without the exception handler. In $FFail$ the instance a_i is removed from the set of instances in use and the exception handler is executed. The guard is identical in both events modelling that any remote procedure call can fail.

5.2.2 Notifications

Failure of grid service instances can cause notifications to never be sent or a failure of network connections can cause notification messages to get lost. In order to detect this we introduce a timeout exception that occurs when a notification has not arrived within the desired time. We do not specify the time explicitly in B but we model the timeout with an event that is chosen non-deterministically for execution. The exact time is a detail that is considered later during the implementation phase.

The grammar of the notification handler substitution is modified in order to incorporate the timeout mechanism.

```
Notif_handler::=''NOTIFICATION'' Name
                 ''SOURCE'' Name ''::'' Name
                 ''THEN'' NG_Substitution
                 ''TIMEOUT'' NG_Substitution
                 ''END''
```

The *notification*-part gives the name of the notification and the *source*-part gives the source of the notification as *<instance>:<grid service machine>*. When a notification is received the non-guarded substitution in the *then*-part

is executed. The *timeout*-handler denotes the substitution that is executed when a notification does not arrive within the desired time.

A notification handler for notification N_1 from instance *inst* of $\mathcal{A}$ can be defined as follows.

NOTIFICATION_HANDLERS
$N_1 Handler \hat{=}$
  NOTIFICATION N_1
  SOURCE $inst \in A$
  THEN $T_{N1}(inst, w, a_i, a_j)$
  TIMEOUT $T_t(inst, w, a_i, a_j)$
  END

$N_1 HandlerOk \hat{=}$
  ANY *inst* **WHERE**
    $inst \in A_Insts \wedge$
    $A_notif(inst) = TRUE \wedge$
    $\neg G_A(x(inst)) \wedge Q_1(x(inst))$
  THEN
    $T_{N1}(inst, w, a_i, a_j) \parallel$
    $A_notif(inst) := FALSE$
  END ;
$N_1 HandlerFail \hat{=}$
  ANY *inst* **WHERE**
    $inst \in A_Insts \wedge$
    $A_notif(inst) = TRUE \wedge$
  THEN
    $A_Destroy(inst);$
    $A_notif := \{inst\} \lhd A_notif;$
    $T_t(inst, w, a_i, a_j)$
  END

As for remote procedure calls, the notification handler $N_1 Handler$ is translated to two different operations in B, $N_1 HandlerOk$ and $N_1 HandlerFail$. The notification handler is a new event and hence, both translated operations have to refine *skip*. The operation, $N_1 HandlerOk$, modelling successful notification handling is translated as in previous version of Distributed B presented in Subsection 3.3. Failed delivery of a notification is modelled by the execution of the timeout handler, $N_1 HandlerFail$. Since it is not known if all events in the grid service machine instance *inst* of $\mathcal{A}$ have become disabled before the failure, only $A_notif(inst) = TRUE$ is present in the guard. The instance *inst* is also here removed from the set of instances in use.

Note that T_{N1} can contain a remote procedure call, which means that the translation needs to be performed in two steps. First the the notification handler is split into two events as described above and then the event containing the remote procedure call is translated as in Subsubsection 5.2.1.

5.2.3 Checking if an Instance is Alive

In order to handle orphans we introduce an *is-alive check* in the grid refinement machine. This check is performed in the grid refinement machine for all instances of all referenced grid service machines. If an instance is unavailable an exception is then raised. To handle these exceptions we introduce a new clause *is_alive_handlers*

```
is_alive_handlers ::=”IS_ALIVE_HANDLERS” is_alive_handler+;
is_alive_handler ::= Name “=”
                     “SOURCE” Name “:” Name
```

```
"THEN" NG_Substitution
"END"
```

The handler substitutions for *is-alive check* exceptions consists of two parts; a source given as *<instance>:<grid service machine>* and a non-guarded substitution describing the behaviour of the system when an exception is raised . The handler for grid service machine $\mathcal{A}$ is given below with its translation.

IS_ALIVE_HANDLERS
$IAHandler \hat{=}$
  SOURCE $inst \in \mathcal{A}$
  THEN $T_{ia}(inst, w, a_i, a_j)$
  END
END

$IAHandler \hat{=}$
  ANY $inst$ **WHERE**
    $inst \in A_Insts$
  THEN
    $A_Destroy(inst);$
    $A_notif := \{inst\} \lhd A_notif;$
    $T_{ia}(inst, w, a_i, a_j)$
  END
END

The handler is translated to an *any*-substitution in B. The *any*-substitution models that an exception can be raised at any time for all instances in use. The handler for failed *is-alive checks* is, like the handlers for notifications, a new event that refines *skip*. As for the failed remote procedure calls and notifications we remove the failed instance from the set of used instances and remove all the variables of the instance.

6 Case study

To better illustrate the use of our fault tolerant extensions to Distributed B we provide a small case study. The application in the case study is abstract, but it illustrates how a problem can be decomposed into a distributed grid application. The application we develop computes the value of a function f of two values v_1 and v_2. The function f could, for example, be a complicated matrix computation that takes two matrices as arguments and produces a third matrix as the result. We assume that the computation of $f(v_1, v_2)$ can split up into computations $f_1(v_1)$ and $f_1(v_2)$ and a computation that composes the final result $g(f_1(v_1), f_1(v_2))$. We thus have the invariant $\forall v_1, v_2.(f(v_1, v_2) = g(f_1(v_1), f_1(v_2)))$. First we model the computation abstractly in an Event B specification. We then decompose the problem and introduce grid features in the refinement.

6.1 The abstract specification

The first abstract specification of the application is written in Event B. The specification abstractly models the computation of f and the possible failures that can occur. First we need to introduce a machine $GDEF$ that defines the set of possible values (matrices) V.

MACHINE
$GDEF$
SETS
V
END

The abstract Event B specification $COMPF$ models the application that computes the function f of two values. The function f used in the computation is modelled as a constant total function. The values are given by the variable $values$, where $values(1)$ is the first argument and $values(2)$ is the second. The result of the computation is then stored in variable $result$. We introduce a variable $state$ for modelling the progress of the computation. State $idle$ models that the computation has not yet started and state $working$ that the computation is in progress. When computation is successfully completed it terminates in state $done$, while a failed computation terminates in state $failed$.

SYSTEM
$COMPF$
SEES
$GDEF$
SETS
$STATE = \{idle, working, done, failed\}$
CONSTANTS
f
PROPERTIES
$f \in (V \times V) \rightarrow V$
VARIABLES
$state, values, result$
INVARIANT
$state \in STATE \wedge$
$values \in 1..2 \rightarrow V \wedge$
$result \in V \wedge$
$(state = done \Rightarrow result = f(values(1), values(2))))$
INITIALISATION
$state := idle \parallel$
ANY v_1, v_2 **WHERE** $v_1 \in V \wedge v_2 \in V$
THEN $values := \{1 \mapsto v_1, 2 \mapsto v_2\}$ **END** $\parallel$
$result :\in V$

The abstract specification contains three events. The event $Start$ assigns new values to the variables $values(1)$ and $values(2)$ and initialises the computation. The event $Finish$ computes the function $f(values(1), values(2))$ and stores the result in variable $result$. The computation can fail while it is in progress as modelled by the event $Fail$.

EVENTS
$Start \mathrel{\hat{=}}$
  ANY v_1, v_2 **WHERE**
    $v_1 \in V \wedge v_2 \in V \wedge$
    $state = idle$
  THEN
    $values := \{1 \mapsto v_1, 2 \mapsto v_2\} \parallel$
    $state := working$
  END ;
$Finish \mathrel{\hat{=}}$
  SELECT $state = working$
  THEN
    $result := f(values(1), values(2)) \parallel$
    $state := done$
  END ;
$Fail \mathrel{\hat{=}}$
  SELECT $state = working$
  THEN $state := failed$
  END
END

6.2 The grid service machine

We like to use a distributed grid application to compute the function f and, hence we need to decompose the computation of f. The function f can be considered as the composition of functions g and f_1. To enable decomposition of the computation of f into computation of f_1 and g we first introduce a grid service machine that computes the function f_1 of a value. The function f_1 is, as function f, modelled as a constant total function. The client can then use instances of this machine for the partial computation. The grid service machine is translated to a B machine for verification. Throughout this subsection, the grid service machine is presented in the left column and its translation is given to the right as in previous sections.

The grid service machine $RF1$ contains a set giving the possible states the service can be in. The translation also contains an automatically generated set and constant for modelling the grid service instances.

GRID_SERVICE
  $RF1$
SEES
  $GDEF$
SETS
  $RS_STATE =$
    $\{f1_init, f1_start, f1_done\}$
CONSTANTS
  f_1
PROPERTIES
  $f_1 \in V \rightarrow V$

MACHINE
  $RF1_V$
SEES
  $GDEF$
SETS
  $RS_STATE =$
    $\{f1_init, f1_start, f1_done\};$
  $RF1_INSTS$
CONSTANTS
  $f_1, RF1_null$
PROPERTIES
  $f_1 \in V \rightarrow V \wedge$
  $RF_null \in RF1_INSTS$

We also introduce variables $f1_state$, $f1_value$ and $f1_result$. The variable $f1_state$ gives the progress of the computation. The value to use in the computation of f_1 is given by the variable $f1_value$. The result is stored

in variable $f1_result$. The invariant states that the variable $f1_result$ contains the result of the computation when $f1_state$ has the value $f1_done$.

```
VARIABLES
  f1_state, f1_value, f1_result
INVARIANT
   f1_state ∈ RS_STATE∧
   f1_value ∈ V∧
   f1_result ∈ V∧
   (f1_state = f1_done ⇒
   f1_result = f1(f1_value))
INITIALISATION
   f1_state := f1_init ||
   f1_value :∈ V ||
   f1_result :∈ V
```

```
VARIABLES
  f1_state, f1_value, f1_result
  RF1_Insts
INVARIANT
   RF1_Insts ⊆ RF1_INSTS∧
   RF1_null ∉ RF1_Insts∧
   f1_state ∈ RF1_Insts → RS_STATE∧
   f1_value ∈ RF1_Insts → V∧
   f1_result ∈ RF1_Insts → V∧
   ∀xx.(xx ∈ RF1_Insts ⇒
   (f1_state(xx) = f1_done ⇒
   f1_result(xx) = f1(f1_value(xx))))
   ...
INITIALISATION
   f1_state := ∅ ||
   f1_value := ∅ ||
   f1_result := ∅ ||
   RF1_Insts := ∅
```

The remote procedure $StartF1$ initiates the computation by assigning the variable $f1_value$ to the value supplied by the client and by setting the state, $f1_state$, to $f1_start$. The client can obtain the result of the computation with a call to the automatically generated read-only remote procedure $GetF1_result$.

```
REMOTE_PROCEDURES
StartF1(v1) ≙
   PRE v1 ∈ V
   THEN
      f1_value := v1||
      f1_state := f1_start
   END
```

```
OPERATIONS
StartF1(inst, v1) ≙
   PRE inst ∈ RF1_Insts ∧ v1 ∈ V
   THEN
      f1_value(inst := v1 ||
      f1_state(inst) := f1_start
   END ;
xx ← GetF1_state(inst) ≙
   PRE inst ∈ RF1_Insts
   THEN xx := f1_state(inst)
   END ;
xx ← GetF1_value(inst) ≙ ...
xx ← GetF1_result(inst) ≙ ...
```

Grid service machine $RF1$ contains one event, $Comp$, that performs the computation of f_1. The event is only executed once and it disables itself by setting the state to $f1_done$.

```
EVENTS
Comp ≙
   SELECT f1_state = f1_start
   THEN
      f1_result := f1(f1_value) ||
      f1_state := f1_done
   END
```

```
Comp ≙
   ANY inst WHERE
    inst ∈ RF1_Insts
   THEN
      SELECT f1_state(inst) = f1_start
      THEN
         f1_result(inst) :=
           f1(f1_value(inst)) ||
         f1_state(inst) := f1_done
      END
   END ;
```

Two different notifications can be sent by this grid service machine. The notification $InitNotification$ is sent when the grid service instance has been initialised properly. The second notification $DoneNotification$, is sent when the computation has finished, i.e., when event $Comp$ has become disabled.

```
NOTIFICATIONS
InitNotification ≙
 GUARANTEES
  f1_state = f1_init
 END ;
DoneNotification ≙
 GUARANTEES
    f1_state = f1_done∧
    f1_result = f1(f1_value)
 END
END
```

The constructor of instances $RF1_GetNew$ and destructor $RF1_Destroy$ are automatically generated during the translation.

```
xx ← RF1_GetNew ≙
   ANY inst WHERE
      inst ∈ RF1_INSTS − RF1_Insts∧
      inst ≠ RF1_null∧
      RF1_Insts ≠ RF1_INSTS − {RF1_null}
   THEN
      RF1_Insts := RF1_Insts ∪ {inst} ||
      f1_state(inst) := f1_init ||
      ANY v_i WHERE v_i ∈ V
      THEN
         f1_value(inst) := v_i ||
         f1_result(inst) := v_i
      END ||
      xx := inst
   END ;
RF1_Destroy(inst) ≙
   PRE inst ∈ RF1_INSTS
   THEN
      RF1_Insts := RF1_Insts − {inst} ||
      f1_state := {inst} ⩤ f1_state ||
      f1_value := {inst} ⩤ f1_value ||
      f1_result := {inst} ⩤ f1_result ||
   END
END
```

The constructor initialises the new instance to the values given in the *initialisations*-clause of the grid service machine and returns it to the client. The destructor removes the instance given as a parameter from the instances in use.

6.3 The grid refinement

The grid refinement machine $COMP_FT1$ is a refinement of the abstract specification $COMPF$ presented in Subsection 6.1. The computation of $f(value(1), value(2))$ in one step is here refined by the distributed computation $g(f_1(value(1)), f_1(value(2)))$. The function g is modelled with a constant total function with the appropriate properties. Instances of the

grid service $RF1$ in Subsection 3.2 are used to compute partial results, $f_1(values(1))$ and $f_1(values(2))$, which is then composed into the final result, $g(f_1(values(1)), f_1(values(2)))$. The grid refinement is translated to a B refinement machine for verification. As with the grid service machine the grid refinement is given in the left column and its translation is given to the right throughout this subsection.

In this refinement step we introduce variables $f1_instances$, $f1_results$ and $tries$.

GRID_REFINEMENT
$COMP_FT1$
REFINES
$COMPF$
SEES
$GDEF$
REFERENCES
$RF1$
CONSTANTS
$MaxTimes, g$
PROPERTIES
$g \in (V \times V) \rightarrow V \wedge$
$\forall (v_1, v_2).(v_1 \in V \wedge v_2 \in V \Rightarrow$
$f(v_1, v_2) = g(f_1(v_1), f_1(v_2))) \wedge$
$MaxTimes \in \mathbb{N}$
VARIABLES
$state, values, result,$
$f1_instances, f1_results, tries$
INVARIANT
$f1_instances \in 1..2 \rightarrow RF1 \wedge$
$f1_results \in RF1 \nrightarrow V \wedge$
$tries \in \mathbb{N} \wedge$
$\ldots$

REFINEMENT
$COMP_FT1_V$
REFINES
$COMPF$
SEES
$GDEF$
INCLUDES
$RF1_V$
PROMOTES
$Comp$
CONSTANTS
$MaxTimes$
PROPERTIES
$g \in (V \times V) \rightarrow V \wedge$
$\forall (v_1, v_2).(v_1 \in V \wedge v_2 \in V \Rightarrow$
$f(v_1, v_2) = g(f_1(v_1), f_1(v_2))) \wedge$
$MaxTimes \in \mathbb{N}$
VARIABLES
$state, values, result,$
$f1_instances, f1_results, tries,$
$RF1_notif$
INVARIANT
$f1_instances \in 1..2 \rightarrow RF1_INSTS \wedge$
$ran(f1_instances) \subseteq$
$(RF1_Insts \cup \{RF1_null\}) \wedge$
$f1_results \in RF1_INSTS \nrightarrow V \wedge$
$dom(f1_results) \subseteq$
$(RF1_Insts \cup \{RF1_null\}) \wedge$
$tries \in \mathbb{N} \wedge$
$\ldots$
$RF1_notif \in RF1_Insts \rightarrow BOOL$

The variable $f1_instances$ is an array containing the two instances that will compute the function values $f_1(values(1))$ and $f_1(values(2))$. The variable $f1_results$ is a partial function from instance to the result obtained from it. The number of times the application has unsuccessfully tried to use instances is modelled by the variable $tries$. The application can fail to use instances $MaxTimes$ number of times before it enters state $failed$ and aborts the computation. Initially, there has been no tries to use instances, $tries = 0$.

The instances, $f1_instances$, are initialised to empty instances, RF_null. Since no results have been obtained the variable $f1_results$ is initialised to the empty set.

INITIALISATION
$state := idle \parallel$
ANY v_1, v_2 **WHERE** $v_1 \in V \wedge v_2 \in V$
THEN $values := \{1 \mapsto v_1, 2 \mapsto v_2\}$ **END** $\parallel$
$result :\in V \parallel$
$f1_instances := (1..2) \times \{RF1_null\} \parallel$
$f1_results := \varnothing \parallel$
$tries := 0$

INITIALISATION
$state := idle \parallel$
ANY v_1, v_2 **WHERE** $v_1 \in V \wedge v_2 \in V$
THEN $values := \{1 \mapsto v_1, 2 \mapsto v_2\}$ **END** $\parallel$
$result :\in V \parallel$
$f1_instances := (1..2) \times \{RF1_null\} \parallel$
$f1_results := \varnothing \parallel$
$tries := 0 \parallel$
$RF1_notif := \varnothing$

The event $GetInstance$ performs an allocation to obtain a new instance of grid service machine $RF1$ to use. If a new instance could not be allocated the variable $tries$ is incremented.

EVENTS
$GetInstance \mathrel{\hat{=}}$
ANY xx**WHERE**
$xx \in 1..2 \wedge$
$state = working \wedge$
$f1_instances(xx) = RF1_null \wedge$
$tries \leq MaxTimes$
THEN
CALL $f1_instances(xx) \leftarrow RF1_GetNew$
EXCEPTION
$f1_instances(xx) := RF1_null \parallel$
$tries := tries + 1$
END
END ;

OPERATIONS
$GetInstanceOk \mathrel{\hat{=}}$
ANY xx**WHERE**
$xx \in 1..2 \wedge$
$state = working \wedge$
$f1_instances(xx) = RF1_null \wedge$
$tries \leq MaxTimes$
THEN
$f1_instances(xx) \leftarrow RF1_GetNew;$
$RF1_notif(f1_instances(xx)) := TRUE$
END ;
$GetInstanceFail \mathrel{\hat{=}}$
ANY xx**WHERE**
$xx \in 1..2 \wedge$
$state = working \wedge$
$f1_instances(xx) = RF1_null \wedge$
$tries \leq MaxTimes$
THEN
$RF1_Destroy(f1_instances(xx));$
$RF1_notif := \{f1_instances(xx)\}$
$\lhd RF1_notif;$
$f1_instances(xx) := RF1_null \parallel$
$tries := tries + 1$
END ;

Note that in the translation of the exception handler we actually remove $RF1_null$ (the value of $f1_instances(xx)$) from the set of instances in use, $RF1_Insts$. Since $RF1_null$ is not in the set, the destructor will have no effect.

Event $Submit$ starts the computation of f_1 of values $values(1)$ and $values(2)$. Both the instances that will be used for the computation need to be different from $RF1_null$. When a failure of an instance is detected the corresponding variable is assigned the value $RF1_null$ since the instance can no longer be used.

```
Submit ≙
 ANY xx WHERE
   xx : 1..2∧
   state = working∧
   f1_instances(1) ≠ RF1_null∧
   f1_instances(2) ≠ RF1_null∧
   f1_instances(xx) ∉ dom(f1_results)
 THEN
   CALL f1_instances(xx).StartF1(values(xx))
   EXCEPTION
     f1_instances(xx) := RF1_null ||
     tries := tries + 1
   END
 END ;
```

```
SubmitOk ≙
 ANY xx WHERE
   xx : 1..2∧
   state = working∧
   f1_instances(1) ≠ RF1_null∧
   f1_instances(2) ≠ RF1_null∧
   f1_instances(xx) ∉ dom(f1_results)
 THEN
   StartF1(f1_instances(xx), values(xx));
   RF1_notif(f1_instances(xx)) := TRUE
 END ;
```

```
SubmitFail ≙
 ANY xx WHERE
   xx : 1..2∧
   state = working∧
   f1_instances(1) ≠ RF1_null∧
   f1_instances(2) ≠ RF1_null∧
   f1_instances(xx) ∉ dom(f1_results)
 THEN
   RF1_Destroy(f1_instances(xx));
   RF1_notif := {f1_instances(xx)}
     ⩤RF1_notif;
   f1_instances(xx) := RF1_null ||
   tries := tries + 1
 END ;
```

The event modelling failure, $Fail$, takes the application to state $failed$ when it has unsuccessfully used instances more than $MaxTimes$ number of times. Successful execution of the application is modelled by the execution of the event $Finish$. This event is enabled when the result from both instances has been obtained via the execution of the notification handlers.

```
Fail ≙
  SELECT state = working∧
    tries > MaxTimes
  THEN state := failed
  END ;
Finish ≙
  SELECT
    state = working∧
    dom(f1_results) = ran(f1_instances)
  THEN
    result := g(f1_results(f1_instances(1)),
      f1_results(f1_instances(2))) ||
    state := done
  END
```

```
Fail ≙
  SELECT state = working∧
    tries > MaxTimes
  THEN state := failed
  END ;
Finish ≙
  SELECT
    state = working∧
    dom(f1_results) = ran(f1_instances)
  THEN
    result := g(f1_results(f1_instances(1)),
      f1_results(f1_instances(2))) ||
    state := done
  END ;
```

If a notification is received successfully the result in the corresponding instance is obtained with a call to the automatically generated remote procedure $GetF1_result$. When a timeout occurs the application stops using the instance. Since $GetF1_result$ is a remote procedure the translation is performed in two steps. The first step splits the notification handler into two events, one for successful handling the notification and the other for timeout handling. The event for successful handling of notifications is then split into two parts to consider both successful execution of the remote procedure $GetF1_result$ as well as the failed execution.

```
NOTIFICATION_HANDLERS
F1DoneNotifHandler ≙
  NOTIFICATION DoneNotif
  SOURCE inst ∈ RF1
  THEN
    IF inst ∈ ran(f1_instances)
    THEN
      VAR xx, ok IN
        ok := TRUE; xx :: VV;
        CALL xx ← inst.GetF1_result
        EXCEPTION
          ok := FALSE
          f1_instances := f1_instances⩤
            {i, v|i ∈ 1..2∧
            inst = f1_instances(i)∧
            v = RF1_null} ||
          f1_results := {inst}
            ⩥f1_results ||
          tries := tries + 1
        END ;
        IF ok = TRUE
        THEN f1_results(inst) := xx
        END
    END
  TIMEOUT
    f1_instances := f1_instances⩤
      {i, v|i ∈ 1..2 ∧ inst = f1_instances(i)∧
      v = RF1_null} ||
    f1_results := {inst}
      ⩥f1_results ||
    tries := tries + 1
  END
```

```
F1DoneNotifHandlerOkOk ≙
  ANY inst WHERE
    inst ∈ RF1_Insts∧
    RF1_notif(inst) = TRUE∧
    ¬(f1_state(inst) = f1_start)∧
    (f1_state(inst) = f1_done∧
    f1_result(inst) = f1(f1_value(inst)))
  THEN
    RF1_notif(inst) := FALSE;
    IF inst ∈ ran(f1_instances)
    THEN
      VAR xx, ok IN
        ok := TRUE; xx :: V;
        xx ← GetF1_result(inst);
        IF ok = TRUE
        THEN f1_results(inst) := xx
        END
      END
    END
  END ;
F1DoneNotifHandlerOkFail ≙
  ANY inst WHERE
    inst ∈ RF1_Insts∧
    RF1_notif(inst) = TRUE∧
    ¬(f1_state(inst) = f1_start)∧
    (f1_state(inst) = f1_done∧
    f1_result(inst) = f1(f1_value(inst)))
  THEN
    RF1_notif(inst) := FALSE;
    IF inst ∈ ran(f1_instances)
    THEN
      VAR xx, ok IN
        ok := TRUE; xx :: V;
        RF1_Destroy(f1_instances(xx));
        RF1_notif := {f1_instances(xx)}
          ⩥RF1_notif;
        ok := FALSE;
        f1_instances := f1_instances⩤
          {i, v|i ∈ 1..2∧
          inst = f1_instances(i)∧
          v = RF1_null} ||
        f1_results := {inst}
          ⩥f1_results ||
        tries := tries + 1
      END ;
      IF ok = TRUE
      THEN f1_results(inst) := xx
      END
    END
  END ;
F1DoneNotifHandlerFail ≙
  ANY inst WHERE
    inst ∈ RF1_Insts∧
    RF1_notif(inst) = TRUE
  THEN
    RF1_Destroy(inst);
    RF1_notif := {inst} ⩥ RF1_notif;
    f1_instances := f1_instances⩤
      {i, v|i ∈ 1..2 ∧ inst = f1_instances(i)∧
      v = RF1_null} ||
    f1_results := {inst}
      ⩥f1_results ||
    tries := tries + 1
  END ;
```

The variable giving the instance that failed is also here assigned the value $RF1_null$, the pending notification is removed and variable *tries* is incremented. Note that we do not introduce an assignment $RF1_notif(inst) := TRUE$ after the call to $GetF1_result$ since it is a automatically defined read-only remote procedure.

Successful *is-alive checks* are not modelled in Distributed B since they do not affect the behaviour of the application. The handling of failed *is-alive checks* is similar to handling notification timeouts. Like for notification timeouts the application stops using the instance that failed.

```
IS_ALIVE_HANDLERS
IAHandler ≙
 SOURCE inst ∈ RF1
 THEN
  f1_instances := f1_instances⩤
   {i, v|i ∈ 1..2 ∧ inst = f1_instances(i)∧
   v = RF1_null} ||
  f1_results := {inst}
   ⩤f1_results ||
  tries := tries + 1
 END
END
```

```
IAHandler ≙
 ANY inst WHERE
  inst ∈ RF1_Insts
 THEN
  RF1_Destroy(inst);
  RF1_notif := {inst} << |RF1_notif;
  f1_instances := f1_instances⩤
   {i, v|i ∈ 1..2 ∧ inst = f1_instances(i)∧
   v = RF1_null} ||
  f1_results := {inst}
   ⩤f1_results ||
  tries := tries + 1
 END
END
```

We have now decomposed the computation of $f(values(1), values(2))$ into a fault tolerant grid application. The grid service machine $RF1$ and the grid refinement $COMP_FT1$ can now be further refined using grid refinement until executable code can be generated. The different components in the development can be proved correct using the tool support for B.

7 Implementation in Java

The grid application development in fault tolerant Distributed B continues until all the non-determinism has been removed and all the used constructs can be implemented, i.e., they belong to the implementable subset of the B language, B0. When all substitutions of the system belong to the B0 language, they can be automatically translated to Java [15]. The event system composed of all events in the *events*-clause can be automatically translated to a *while*-loop in Java, if all the guards Gi and the substitutions Si belong to the B0 language.

```
E1 ≙
WHEN G1
THEN S1 END
...
En ≙
WHEN Gn
THEN Sn END
```

```
while (true) {
   if(G1) S1;
   ...
   else if(Gn) Sn;
   else <stop loop>;
}
```

The place holder <*stop loop*> consists of statements that terminate the loop

in the main program and statements for sending notifications and pausing the execution in implementations of grid service machines.

In the client the grid service instances are translated to objects encapsulating the grid specific features. These features include instance allocation, remote procedure calls, notification handling, notification timeouts and management of *is-alive checks*. In the implementation of grid service machines an additional layer encapsulating grid features such as state management and *is-alive timeouts* is inserted between the Globus toolkit and the translated grid service machine. The code for the objects and the additional layer need to be manually created once, but can then be reused in all Distributed B applications.

Java code is generated for each concrete grid refinement machine that is a refinement of a grid service machine and for the root grid refinement machine that is a refinement of an Event B model. When the code has been generated we have implemented the grid application in a formal manner, where we can show that the implementation is a correct refinement of the abstract specification.

8 Conclusions

We have earlier developed a language Distributed B [5] that extends Event B for designing and implementing correct grid applications. In that paper we introduced two new types of machines, *grid service machine* and *grid refinement machine,* for handling grid specific issues in Event B. In this paper we have modified Distributed B for developing fault tolerant and robust grid applications. We introduced exception handling to take care of exceptions raised due to failed remote procedure calls, as well as timeouts to discover lost notifications. In order to handle orphan grid service instances we introduced a special checking mechanism and facilities to handle exceptions raised by it. These new features force the developer to consider the fault tolerance of grid applications throughout the development and enables formal proofs of correctness. Hence, we have proposed a method for implementing fault tolerant grid applications where the implementation can be proved correct with respect to its specification.

There are several middlewares comparable to Globus toolkit that support advanced fault tolerance mechanisms. For example, fault tolerance in CORBA [10, 14] is based on replication. In CORBA replicas of an object form an object group. Object groups then provide replication transparency and failure transparency. Replication is complementary to the mechanisms presented here and both can be used together for developing very reliable grid applications. Fault detection is an important part of a fault tolerant application. A service for detecting faulty components in a grid environment using *unreliable fault detectors* is presented in a paper by Stelling et al. [13]. However, the service is no longer part of the Globus toolkit due to a number of deficiencies, such as excessive resource usage and difficulties in supporting

it.

From a performance perspective our fault tolerant language for developing grid applications could be improved. One of the most challenging problems when constructing fault tolerant grid systems is to handle the orphan grid service instances. Our approach uses timers and extra remote procedure calls, which might not be optimal. The handling of orphans could be improved as discussed in detail by Panzieri and Shrivastava [12] and by Tanenbaum and Steen [14]. Furthermore, the fault tolerance mechanism will immediately delete references to grid service instances that experiences problems. This can lead to the deletion of an entire subtree of instances for one failure. Hence, more efficient mechanisms that would maintain the partial results should be investigated.

The language we proposed in this paper provides a convenient formal development process for fault tolerant grid applications. The applications will by construction have an architecture that is fault tolerant and implementable. Furthermore, the applications are modelled in terms of grid primitives with a precise meaning and the specifications of them will therefore be clear to understand. Our approach to adapt Event B to the Globus Toolkit middleware is not limited to that specific middleware, but it can be applied to other middlewares for distributed systems as well.

References

[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] J. R. Abrial, D. Cansell and D. Méry. Refinement and Reachability in Event B. In H. Treharne et al, editors, *proceedings of the 4th international conference of Z and B users: ZB2005.* LNCS 3455, Guildford, UK, pp. 144-163, Springer-Verlag, 2005.

[3] J. R. Abrial and L. Mussat. Event B Reference Manual, 2001. `http://www.atelierb.societe.com/ressources/evt2b/eventb_reference_manual.pdf`. (accessed 10.08.2005)

[4] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, pp. 131-142, 1983.

[5] P. Boström and M. Walden. An extension of Event B for developing grid systems. In H. Treharne et al, editors, *proceedings of the 4th international conference of Z and B users: ZB2005.* LNCS 3455, Guildford, UK, pp. 144-163, Springer-Verlag, 2005.

[6] K. Czajkowski, et. al. Open Grid Services Infrastructure, 2003. `http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf`. (accessed 10.08.2005)

[7] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.

[8] I. Foster, C. Kesselman and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *The International Journal of Supercomputer Applications,* 15(3), 2001.

[9] I. Foster, C. Kesselman, J. Nick and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, 2002.
`http://www.globus.org/alliance/publications/papers/ogsa.pdf`.
(accessed 10.08.2005)

[10] Object Management Group. Fault tolerant CORBA, 2001,
`http://www.omg.org/docs/formal/01-09-29.pdf`.
(accessed 10.08.2005)

[11] The Globus Alliance. Globus Toolkit. 2005. `http://www.globus.org/`.
(accessed 10.08.2005)

[12] F. Panzieri and S. K. Shrivastava. Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing. *IEEE Transactions on Software Engineering*, 14(1), pp 30-37, 1988.

[13] P. Stelling, C. DeMatteis, I. Foster, C. Kesselman, C. Lee, G. von Laszewski. A Fault Detection Service for Wide Area Distributed Computations, *Cluster Computing*, 2, pp. 117-128, 1999

[14] A. S. Tanenbaum and M. Van Steen. *Distributed systems principles and paradigms.* Prentice Hall. 2002

[15] J. C. Voisinet, B. Tatibouet and A. Hammand. JBTools: An experimental platform for the formal B method. In *Proceedings of the inaugural conference on the Principles and Practice of programming and Proceedings of the second workshop on Intermediate representation engineering for virtual machines.* National University of Ireland, 2002

[16] M. Waldén and K. Sere. Reasoning About Action Systems Using the B-Method. *Formal Methods in Systems Design*, 13:5-35, 1998.

A A fault tolerant grid refinement machine

GRID_REFINEMENT
$\mathcal{C}_{FT}$
REFINES
$\mathcal{C}$
REFERENCES
$\mathcal{A}$
VARIABLES
$v, w, a_1, \ldots, a_n$
INVARIANT
$a_1 \in \mathcal{A} \wedge$
$\ldots a_n \in \mathcal{A} \wedge$
$J(v, w, a_1, \ldots, a_n)$
INITIALISATION
$a_1 := A_null \parallel$
$\ldots$
$Init'(v, w)$
EVENTS
$E_1 \hat{=}$
ANY y **WHERE**
$G'_1(v, y)$
THEN
$S'_1(v, y);$
CALL $a_i \leftarrow A_GetNew$
EXCEPTION T_e
END
END ;
$E_2 \hat{=}$
WHEN $G'_2(v, w)$
THEN $S'_2(v, w)$
END ;
$F \hat{=}$
WHEN $H(v, w, a_i)$
THEN
$S_n(w) \parallel$
CALL $a_i.Proc(f(v, w))$
EXCEPTION $T_f(w, a_i)$
END
END
NOTIFICATION_HANDLERS
$N_1Handler \hat{=}$
NOTIFICATION N_1
SOURCE $inst \in A$
THEN $T_{N1}(inst, w, a_i, a_j)$
TIMEOUT $T_t(inst, w, a_i, a_j)$
END
IS_ALIVE_HANDLERS
$IAHandler \hat{=}$
SOURCE $inst \in \mathcal{A}$
THEN $T_{ia}(inst, w, a_i, a_j)$
END

END

REFINEMENT
$\mathcal{C}_{FTV}$
REFINES
$\mathcal{C}$
INCLUDES
$\mathcal{A}_V$
PROMOTES
E_A
VARIABLES
$v, w, a_1, \ldots, a_n, A_notif$
INVARIANT
$a_1 \in A_INSTS \wedge$
$a_1 \in A_Insts \cup \{A_null\} \wedge$
$\ldots$
$J_V(v, w, a_1, \ldots, a_n) \wedge$
$A_notif \in A_Insts \rightarrow BOOL$
INITIALISATION
$a_1 := A_null \parallel$
$\ldots$
$Init'(v, w) \parallel$
$A_notif := \varnothing$
OPERATIONS
$E_1Ok \hat{=}$
ANY y **WHERE**
$G'_1(v, y)$
THEN
$S'_1(v, y);$
$a_i \leftarrow A_GetNew;$
$A_notif(a_i) := TRUE$
END ;
$E_1Fail \hat{=} \ldots$
$E_2 \hat{=} \ldots$
$FOk \hat{=}$
WHEN $H(v, w, a_i)$
THEN
$S_n(w) \parallel$
$Proc(a_i, f(v, w));$
$A_notif(a_i) := TRUE$
END ;
$FFail \hat{=} \ldots$
$N_1HandlerOk \hat{=}$
ANY $inst$ **WHERE**
$inst \in A_Insts \wedge$
$A_notif(inst) = TRUE \wedge$
$\neg G_A(x(inst)) \wedge Q_1(x(inst))$
THEN
$T_{N1}(inst, w, a_i, a_j) \parallel$
$A_notif(inst) := FALSE$
END ;
$N_1HandlerFail \hat{=} \ldots$
$IAHandler \hat{=}$
ANY $inst$ **WHERE**
$inst \in A_Insts$
THEN
$A_Destroy(inst);$
$A_notif := \{inst\} << |A_notif;$
$T_{ia}(inst, w, a_i, a_j)$
END
END

Turku

Centre *for*

Computer

Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 952-12-1594-1
ISSN 1239-1891