Pontus Boström | Marina Waldén

An extension of Event B for developing grid systems

Turku Centre *for* Computer Science

TUCS Technical Report
No 632, November 2004

An extension of Event B for developing grid systems

Pontus Boström
Åbo Akademi University, Department of Computer Science/TUCS
Lemminkäisenkatu 14 A, 20520 Turku, Finland
pontus.bostrom@abo.fi
Marina Waldén
Åbo Akademi University, Department of Computer Science/TUCS
Lemminkäisenkatu 14 A, 20520 Turku, Finland
marina.walden@abo.fi

TUCS Technical Report
No 632, November 2004

Abstract

Computational grids have become widespread in organizations for handling their need for computational resources and the vast amount of available information. These grid systems as other distributed systems are often complex and formal reasoning about them is needed, in order to ensure their correctness and to structure their development. Event B is a formal method with tool support that is meant for stepwise development of distributed systems. To facilitate the implementation of grid systems we here propose extensions to Event B that take grid specific features into account. We add new constructs to model the client-server architecture of grid systems, as well as important features like communication and synchronisation. We introduce the extensions in such a manner that the necessary proof obligations are automatically generated and the system can be directly implemented.

Keywords: Grid Systems, Distributed Systems, Event B, Language Extensions, Implementation

TUCS Laboratory
Distributed Systems Design Laboratory

1 Introduction

Organizations need the ability to efficiently utilise existing hardware and be able to effectively share information with each other. Computational grids have become a popular approach to enable organizations to handle the vast amount of available information. These grids are also used for solving problems in, e.g., biology, nuclear physics and engineering. Grid computing [9, 14] is a distributed computing paradigm that differ from traditional distributed computing in that it is aimed toward large scale systems that even span organizational boundaries.

The development of correct grid systems is difficult with traditional software development methods. Hence, formal methods are needed in order to ensure their correctness and structure their development from specification to implementation. The Action Systems formalism [5] is a formal method that is well suited for developing large distributed 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 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 [20] and Event B [3]. B Action Systems models Action Systems in the B Method, while Event B also extends original B with new constructs. We mainly use Event B in this paper.

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. The problem becomes especially apparent when developing distributed systems with complicated synchronization and communication patterns. Therefore, we propose new extensions to Event B in order to be able to construct models of grid systems that can be implemented and to verify their correctness in a convenient way. The language obtained by the extensions will be referred to as Distributed B in the rest of the paper.

The language Distributed B is targeted towards Grid systems using the Globus Toolkit [11] middleware. Grid systems usually have a client-server architecture. This means that there is a client that initiates communication with the server, which only responds to the clients' requests. Distributed B supports client-server architectures with multiple concurrent accesses by the same client to several servers. The main communication mechanism of the grid middleware is remote procedure calls. However, the grid middleware also supports asynchronous notifications sent from a server to a client. Both these communication primitives are used in Distributed B. The constructs are introduced in such a manner that they ensure that the system will be implementable and all needed proof obligations can be automatically generated.

In section 2 we describe formal development of systems in Event B. In section 3 we give an overview of the grid technology and discuss how the grid features are incorporated into Event B. The new constructs, grid service machine and grid refinement machine, are presented in sections 4 and 5, respectively. In section 6 we discuss implementation issues and in section 7 we conclude.

2 Formal development with Event B

In order to be able to develop correct grid systems and other distributed systems, we need to reason about these systems in a formal manner. Furthermore, it is important that the formal reasoning is facilitated by good tool support. Action Systems is a well established formalism for reasoning about distributed systems [5]. However, it lacks good tool support. Event B [3] is a formalism that is based on Action Systems and is an extension of the B Method for developing distributed systems. This formalism is also provided with tool support via the B Method. Because of this we have chosen Event B as the formalism within which we develop our framework for specifying and implementing grid systems.

2.1 Abstract specifications

An abstract model of a system within Event B is encapsulated in a *system*-machine and is identified by a unique name. Let us study the abstract model $\mathcal{C}$ below.

SYSTEM $\mathcal{C}$
VARIABLES
  x
INVARIANT
  $I(x)$
INITIALISATION
  $x := x_0$
EVENTS
  $E_1 \ \hat{=} \ S_1;$
  $E_2 \ \hat{=} \ S_2;$
  $\ldots$
END

Each variable x in the *variables*-clause is associated with some domain of values. The set of possible assignments of values to the state variables constitutes the state space. The data invariant $I(x)$ in the *invariant*-clause defines the state space of the variables and their invariant properties. In the *initialisation*-clause initial values are assigned to these variables. The *events*-clause contains events describing the behaviour of the system. Each event in the *events*-clause is a substitution statement, where the substitution, for example, can be a *skip*-substitution, a simple substitution, a multiple substitution, a sequential substitution, a preconditioned substitution, a conditional substitution, a guarded substitution or a nondeterministic guarded substitution. The semantics of these substitution statements is given by the weakest precondition calculus developed by Dijkstra [8].

$$
\begin{array}{lll}
\mathrm{wp}(skip, Q) & = & Q \\
\mathrm{wp}(x := e, Q) & = & Q[x := e] \\
\mathrm{wp}(x := e \;\|\; y := f, Q) & = & Q[x, y := e, f], \text{ where } x \cap y = \emptyset \\
\mathrm{wp}(x := e;\; y := f, Q) & = & (Q[y := f])[x := e] \\
\mathrm{wp}(\textbf{PRE } G \textbf{ THEN } S \textbf{ END}, Q) & = & G \wedge \mathrm{wp}(S, Q) \\
\mathrm{wp}(\textbf{IF } G \textbf{ THEN } S \textbf{ ELSE } T \textbf{ END}, Q) & = & (G \Rightarrow \mathrm{wp}(S, Q)) \wedge (\neg G \Rightarrow \mathrm{wp}(T, Q)) \\
\mathrm{wp}(\textbf{SELECT } G \textbf{ THEN } S \textbf{ END}, Q) & = & G \Rightarrow \mathrm{wp}(S, Q) \\
\mathrm{wp}(\textbf{ANY } x \textbf{ WHERE } G \textbf{ THEN } S \textbf{ END}, Q) & = & \forall x.G \Rightarrow \mathrm{wp}(S, Q)
\end{array}
$$

Here, Q and G are predicates, x and y are variables, e and f are expressions, while S and T are arbitrary substitution statements.

An event is considered to consist of a guard and a body. For example, for event $E \mathrel{\hat{=}}$ **SELECT** G **THEN** S **END** the guard, $\mathrm{gd}(E)$, is $(G \wedge \mathrm{gd}(S))$. When the guard of an event evaluates to *true* in a given state, the event is said to be enabled. Only enabled events are considered for execution. If several events are enabled, they are executed in random order. Events that do not share variables can be executed in parallel. When there are no enabled, events the system terminates. The events are considered to be atomic and, hence, only their input-output behaviour is of interest.

In grid systems remote procedures play an important role. Remote procedures [18] are, however, not supported in Event B. The reason for this is that a model in Event B is closed, i.e., the system is modeled as a whole without relying on outside information. For reasoning about remote procedures we rely on the formalism B Action Systems [20] , another formalism applying Action Systems within the B Method and related to Event B. Remote procedures are discussed in more detail elsewhere [18, 7].

2.2 Decomposing event systems

Grid systems are often very complex systems. Therefore, it is beneficial to split these systems into several smaller ones during the development [6]. Let us study how an event system $\mathcal{C}$ can be decomposed into two components $\mathcal{C}_1$ and $\mathcal{C}_2$. System $\mathcal{C}$ contains the variables x, y and z, where the event E_1 refers to x and z, and event E_2 to y and z. We assume that E_1 does not modify z.

SYSTEM $\mathcal{C}$
VARIABLES
  x, y, z
INVARIANT
  $I_{C1}(x, z) \wedge I_{C2}(y, z)$
INITIALISATION
  $x := x_0 \;||\; y := y_0 \;||\; z := z_0$
EVENTS
  $E_1 \mathrel{\hat{=}}$ **SELECT** G_1 **THEN** S_1 **END**;
  $E_2 \mathrel{\hat{=}}$ **SELECT** G_2 **THEN** S_2 **END**
END

The parallel decomposition of system $\mathcal{C}$ into the components $\mathcal{C}_1$ and $\mathcal{C}_2$ is then defined by splitting the variables and events as follows.

SYSTEM $\mathcal{C}_1$	**SYSTEM** $\mathcal{C}_2$
EXTENDS	
$\mathcal{C}_2$	
VARIABLES	**VARIABLES**
x	y, z
INVARIANT	**INVARIANT**
$I_{C1}(x, z)$	$I_{C2}(y, z)$
INITIALISATION	**INITIALISATION**
$x := x_0$	$y := y_0 \mid\mid z := z_0$
EVENTS	**EVENTS**
$E_1 \ \hat{=}$ **SELECT** G_1 **THEN** S_1 **END**	$E_2 \ \hat{=}$ **SELECT** G_2 **THEN** S_2 **END**
END	**END**

Here we say that system $\mathcal{C}_1$ *extends* system $\mathcal{C}_2$ indicating that $\mathcal{C}_1$ is composed in parallel with $\mathcal{C}_2$. Note that the *extends*-clause is as defined in the original B Method. After the decomposition the variable x is located in $\mathcal{C}_1$ while y and z are in $\mathcal{C}_2$. The invariant, the initialisation, as well as the events referring to the variable x are included in $\mathcal{C}_1$, while the ones referring to y and z are given in $\mathcal{C}_2$. In $\mathcal{C}_2$ the variable z is a global variable, since it is referenced also in system $\mathcal{C}_1$. The decomposition rule can be applied in reverse and is then called parallel composition [6, 7].

When composing event systems we also consider prioritised composition [13]. The prioritised composition $\mathcal{A}//\mathcal{B}$ denotes the parallel composition between the event systems $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{A}$ has a higher priority than $\mathcal{B}$. If an event is enabled in $\mathcal{A}$, it will always be executed before any event in $\mathcal{B}$.

2.3 Refinement

In Event B we can refine an abstract specification in a stepwise manner to a more concrete and detailed specification. New variables can be introduced and the old ones can be refined to more concrete ones. This is reflected in the substitutions of the events, as well. Furthermore, new events that only assign the new variables may be introduced. In a refinement step we can also merge several events into one event, as well as refine one event by several new events.

Let us assume that we have two event systems $\mathcal{C}$ and $\mathcal{C}_1$ as below. The variable x in $\mathcal{C}$ is refined to x' in $\mathcal{C}_1$, while y is the new variable introduced in $\mathcal{C}_1$. The events E_i are refined by the corresponding events E'_i to also take y into account. The events F_j are introduced in this refinement step and refer only to the new variable y.

MACHINE $\mathcal{C}$	**REFINEMENT** $\mathcal{C}_1$
	REFINES $\mathcal{C}$
VARIABLES	**VARIABLES**
x	x', y
INVARIANT	**INVARIANT**
$I(x)$	$J(x, x', y)$
INITIALISATION	**INITIALISATION**
$x := x_0$	$x' := x'_0 \parallel y := y_0$
EVENTS	**EVENTS**
$E_1 \ \hat{=} \ S_1;$	$E'_1 \ \hat{=} \ S'_1;$
...	...
$E_n \ \hat{=} \ S_n$	$E'_n \ \hat{=} \ S'_n;$
END	$F_1 \ \hat{=} \ T_1;$
	...
	$F_m \ \hat{=} \ T_m$
	END

When invariant $J(x, x', y)$ is a relation between the abstract variables x and the concrete variables x' and y, we write $E \sqsubseteq_J E'$ to denote that the abstract event E is data refined by the concrete event E' under invariant J [5]. In order to show that system $\mathcal{C}_1$ is a refinement of $\mathcal{C}$ under invariant J, $\mathcal{C} \sqsubseteq_J \mathcal{C}_1$, the following proof obligations should hold [3]:

1. $Init \sqsubseteq_J Init'$
2. $E_i \sqsubseteq_J E'_i$, for $i \in 1..n$
3. $skip \sqsubseteq_J F_j$, for $j \in 1..m$
4. $J \wedge \neg(\mathrm{gd}(E'_1) \vee \ldots \vee \mathrm{gd}(E'_n) \vee \mathrm{gd}(F_1) \vee \ldots \vee \mathrm{gd}(F_m)) \Rightarrow \neg(\mathrm{gd}(E_1) \vee \ldots \vee \mathrm{gd}(E_n))$
5. $J \Rightarrow V \in \mathbb{N}$
6. $\mathrm{gd}(F_j) \Rightarrow \mathrm{wp}(n := V, \mathrm{wp}(F_j, V < n))$

The initialisation in the refined system maintains the behaviour of the abstract system (1). Every event E_i in the abstract system is refined by an event E'_i in the concrete system (2). New events F_j should only refer to the new variables (3). They should not change the behaviour of the abstract system. The refined event system must not terminate more often than the abstract one (4). The behaviour of the abstract system should be preserved and, hence, the new events should terminate when executed in isolation (5 and 6). Here, V is a variant that is decreased by every new event F_j. All these proof obligations can be automatically generated by the tools for Event B.

For the remote procedures we rely on the proof obligations for B Action Systems [7]. Let us assume that we have procedure P_k in $\mathcal{C}$ that is refined by P'_k in $\mathcal{C}_1$. When considering procedures in event systems the following additional proof obligations should hold.

7. $P_k \sqsubseteq_J P'_k$, for $k \in 1..h$
8. $J \wedge \mathrm{gd}(P_k) \Rightarrow \mathrm{gd}(P'_k)$, for $k \in 1..h$

The abstract remote procedure P_k should be refined by the corresponding procedure P'_k in $\mathcal{C}_1$ (7). Furthermore, the guards of the procedures may not be changed (8). Proof obligation (7) can be automatically generated via Event B, while proof obligation (8) requires some extra constructs corresponding to the ones in [20].

3 Grid systems in Event B

Relying on Event B we can formally specify correct grid systems. However, it is not straightforward to develop the specification in such a manner that it can be directly implemented. We propose an extension of Event B, Distributed B, that enable us to create implementable specifications of grid systems in a convenient way. Let us first study grid systems.

3.1 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) [10] 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) [12] gives a precise technical definition of what a grid service is. The Globus Toolkit 3.x [11], an implementation of the OGSI specification, has become defacto standard toolkit for implementing grid systems. This is also the toolkit we use as grid middleware for Distributed B in this paper.

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. A server is referred to as a grid service in Globus Toolkit, since it provides services to other grid components. 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. By using notifications a grid service can asynchronously notify clients about changes in its state. The state of grid services are 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.

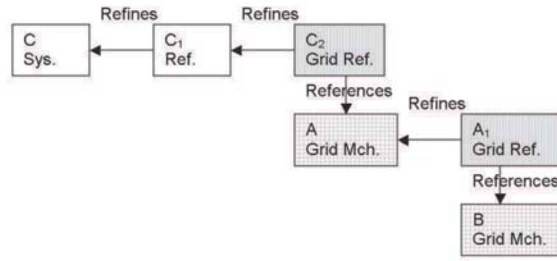

Figure 1: The structure of the Distributed B development

3.2 Extending Event B

The main purpose of the language Distributed B is to be able to specify, verify and implement correct grid systems in a convenient way. As for grid systems the most common communication mechanism in Distributed B is remote procedure calls. However, in order to support concurrent accesses by the same client to multiple grid services, Distributed B also takes into account notifications.

In order to meet the requirements above, we propose to extend Event B with two types of machines, a *grid service machine* modelling abstract grid service features and a *grid refinement machine* for refining an ordinary Event B model by introducing grid features or for refining a grid service machine. A grid service machine is a template of which a client (a grid refinement machine) can obtain instances. Using terminology from object oriented programming, the grid service machine can be viewed as a class and the instances as objects of the class. This new composition mechanism is expressed with the *references* construct in the grid refinement machine. Several instances of the same grid service machine can be controlled by the same client as a master can control several identical worker nodes. A grid service machine contains specifications of remote procedures, events and notifications. The grid refinement machine, on the other hand, has clauses for refined remote procedures and events, as well as a clause for handling notifications. The clients and the grid services use remote procedure calls and notifications to communicate and synchronize with each other. For example, a client can make a requests to a grid service with a remote procedure call and when the request has been carried out a notification is sent back to the client.

The development of the grid system shown in Figure 1 starts with an initial specification, $\mathcal{C}$, in Event B that is refined in a number of steps, $\mathcal{C}_1$. The specification is then split up into a client, $\mathcal{C}_2$, and a number of grid services, $\mathcal{A}$. The grid services can in turn be independently refined further, $\mathcal{A}_1$, and reference new grid services, $\mathcal{D}$. For simplicity we assume that each grid service machine can only be referenced from one grid refinement machine.

Throughout the development of the system the grid constructs are translated to ordinary B machines for verification purposes. Note that we translate the Distributed B specifications to the B Method and not to Event B. The reason for this is

that the current tool support also translate Event B specifications to the B Method for verification. We translate the Event B constructs in Distributed B to the B Method in the same manner as the current tools for Event B.

4 Grid service machines

In Distributed B an abstract model of a grid service is given as the construct *grid service machine*. Grid service machines extend Event B with clauses for specifying remote procedures and notifications. A grid service can wait for a remote procedure call from a client. Upon the call it performs the requested task. When the task has been completed, i.e., when all the events in the grid service machine has become disabled, a notification is sent. By choosing to send the notification only after the task has been completed, the notification mechanism can be implemented using the Globus Toolkit in a straightforward manner.

4.1 Grammar for the grid service machine

The grammar for the grid service machine is an extension of the grammar for an abstract system in Event B. Here only the differences between the grammars are shown.

```
gridservice ::= "GRID_SERVICE" Name
                Clause_gridservice+
Clause_gridservice ::=
   Clause_system_abstract |
   Clause_rpcs |
   Clause_notif
Clause_rpcs ::= "REMOTE_PROCEDURES" Rpc_oper+;
Rpc_oper ::= Header_operation "=" NG_Substitution
Clause_notif ::= "NOTIFICATIONS" Notif+;
Notif::= Name "=" "GUARANTEES" Predicate "END"
```

As in an abstract system in Event B the grid service machine contains constants, sets, variables and predicates on them. The variables are first initialised and then modified by the events. Additionally, the grid service has a number of remote procedures that other services can access. A remote procedure is an implementable operation in the B Method, i.e., it only contains non-guarded substitutions (here called *NG_Substitution*) of the set of substitutions in Event B. The *notifications*-clause contains *guarantees*-statements with conditions indicating when the notifications can be sent to the client. A notification is sent when none of the events in the *events*-clause are enabled and the predicate in its *guarantees*-statement holds.

4.2 Mapping the specification to B.

An abstract grid service machine contains clauses which do not exist in an Event B specification. In order to be able to use tool support for verifying the consistency of the grid service machine , we need to translate the grid service machine to an abstract machine specification in the B Method.

4.2.1 Translation of the grid service machine to B

In a system developed within Distributed B it is assumed that all available instances of all the grid services are created upon initialisation of the system. The index service of Globus Toolkit then provides references to available grid service instances of correct type. In the B Method the set of instances that can be obtained from the index service first has to be defined. This dynamic management of instances of machines are not directly supported in the B Method and, hence, it has to be explicitly modeled [4, 17].

Let us assume that we have a grid service machine $\mathcal{A}$. The set of instances of $\mathcal{A}$ that can be obtained from the index service is then given as the set *A_INSTANCES*. The constant *A_null* models an empty instance of grid service machine $\mathcal{A}$. Upon a request for a new instance from the index service, the value *A_null* is returned when no non-empty instance is available.

SETS
$A_INSTANCES$
CONSTANTS
A_null
PROPERTIES
$A_null \in A_INSTANCES$

The variable *A_Instances* models the set of non-empty instances of $\mathcal{A}$ currently in use by the client. They are obtained dynamically from the index service.

VARIABLES
$A_Instances$
INVARIANT
$A_Instances \subseteq A_INSTANCES \wedge$
$A_null \notin A_Instances$

All the variables in a grid service machine are translated to functions from the set of current instances to the variable types. Assume that grid service machine $\mathcal{A}$ has a variable x with type X. When $\mathcal{A}$ is translated to the B Method the type of x is defined as $x \in A_Instances \rightarrow X$.

When we translate remote procedures to the B Method to take an instance *inst* into account, we introduce the instance for which it is called as an additional parameter. For example, procedure $Proc(p) \hat{=} P$ becomes $Proc(inst,\ p) \hat{=} P(inst)$ upon translation. The events are translated to non-deterministic guarded substitutions (*any*-substitutions) to take instances into account. Hence, the event $E_1 \hat{=} S_1$ in the grid service machine becomes:

$$E_1 \hat{=} \textbf{ANY}\ inst\ \textbf{WHERE}\ inst \in A_Instances\ \textbf{THEN}\ S_1(inst)\ \textbf{END}$$

Since the notifications should be enabled when the events of the grid service machine have become disabled, we add the following predicate to the invariant of the abstract machine upon translation:

$$\forall inst.(inst \in A_Instances \wedge \neg \mathrm{gd}(\mathcal{A}(inst)) \Rightarrow Q_1(inst) \vee ... \vee Q_n(inst))$$

where Q_i is the predicate of the *guarantees*-statement in notification *i* in $\mathcal{A}$. The predicate states that one of the notifications is enabled when all events of $\mathcal{A}$ are disabled.

In order for a client to be able to obtain new instances for a grid service machine via the index service, a procedure *GetNew* is automatically generated in the translated abstract B machine. This procedure can be viewed as the constructor of instances.

```
z ← A_GetNew ≙
CHOICE
    ANY inst WHERE
        A_Instances ≠ A_INSTANCES ∧
        inst ∈ A_INSTANCES − A_Instances ∧ inst ≠ A_null
    THEN
        A_Instances := A_Instances ∪ {inst} ||
        x(inst) := x0 || z := inst
    END
OR z := A_null
END
```

The procedure ensures that the instance returned is not already in use and returns *A_null* if no non-empty instance is available. If variable *x* is assigned x_0 in the *initialisation*-clause of grid service machine $\mathcal{A}$, variable *x* for the returned instance *inst* of $\mathcal{A}$ is assigned x_0, $x(inst) := x_o$, in *A_GetNew.*

Grid services allocated by a client may need to be returned to the index service. Hence, a procedure *Destroy* is automatically generated for each grid service machine upon translation to the B Method to return an instance no longer in use.

```
A_Destroy(inst) ≙
PRE inst ∈ A_INSTANCES
THEN
    IF inst ∈ A_Instances
    THEN
        x := {inst} ⩤ x ||
        A_Instances := A_Instances − {inst}
    END
END
```

The operation *A_Destroy* in $\mathcal{A}$ deletes the instance, *inst*, from the set of instances in use and marks the instance as available in the index service. This procedure can be viewed as the destructor of instances.

4.2.2 Example of a grid service machine

As an example of translating grid service machines in Distributed B to the B Method, let us study grid service machine *ADDER* that computes the sum of all values it receives. The machine has two remote procedures, *SetNewData* and *GetResult*. The new value to be added to the sum is given via procedure *SetNewData*. The result of the latest computation can be obtained via procedure *GetResult*. The variable *sum* gives the current result of the sum computation, while the variable *param* contains the latest value received via *SetNewData*. The variable *state* ensures that all the received values are added once and only once to *sum*. The actual computation of the sum is performed in event *Comp*. A notification is sent after the initialisation, *InitNotif*, as well as after a new sum has been computed, *DoneNotif*.

The grid service machine *ADDER* is translated to the abstract B machine *ADDER_VERIFICATION* for verification as follows:

```
GRID_SERVICE
  ADDER

VARIABLES
  sum, param, state
INVARIANT
  sum ∈ ℕ∧
  param ∈ ℕ∧
  state ∈ STATE

INITIALISATION
  sum := 0 || param := 0 ||
  state := init
REMOTE_PROCEDURES
 SetNewData(p) ≙
   PRE p ∈ ℕ
   THEN
     param := p ||
     state = start
   END ;
z ← GetResult ≙
   BEGIN z := sum
   END

EVENTS
Comp ≙
   SELECT state = start
   THEN
     sum := sum + param ||
     state := done
   END
NOTIFICATIONS
InitNotif ≙
 GUARANTEES state = init END ;
DoneNotif ≙
 GUARANTEES state = done END
END
```

```
MACHINE
 ADDER_VERIFICATION
 ...
VARIABLES
 sum, param, state, ADDER_Instances
INVARIANT
ADDER_Instances ⊆ ADDER_INSTANCES∧
A_null ∉ ADDER_Instances∧
sum ∈ ADDER_Instances → ℕ∧
param ∈ ADDER_Instances → ℕ∧
state ∈ ADDER_Instances → STATE∧
∀inst.(inst ∈ ADDER_Instances∧
   ¬(state(inst) = start) ⇒
   state(inst) = idle ∨ state(inst) = done)
INITIALISATION
 sum := ∅ || param := ∅ || state := ∅ ||
 ADDER_Instances := ∅
OPERATIONS
SetNewData(inst, p) ≙
   PRE p ∈ ℕ ∧ inst ∈ ADDER_Instances
   THEN
     param(inst) := p ||
     state(inst) = start
   END ;
z ← GetResult(inst) ≙
   PRE inst ∈ ADDER_Instances
   THEN z := sum(inst)
   END ;
y ← ADDER_GetNew ≙ ... ;
ADDER_Destroy ≙ ... ;

Comp ≙
  ANY inst WHERE inst ∈ ADDER_Instances
  THEN
    SELECT state(inst) = start
    THEN
      sum(inst) := sum(inst) + param(inst) ||
      state(inst) := done
    END
  END
END
```

The types of the variables in the grid service machine are translated to functions from instances of the grid service machine to data values. For example, the variable *sum* has type ℕ in *ADDER*, while it is a total function from the instances *ADDER_Instances* to ℕ in *ADDER_VERIFICATION*. Instances are created and deleted by the procedures *ADDER_GetNew* and *ADDER_Destroy* introduced in *ADDER_VERIFICATION*. The remote procedures $SetNewData$ and $GetResult$ take the instances into account. An additional parameter is introduced to denote for which instance the procedure is called. Event *Comp* is translated to an *any*-substitution for a non-deterministically chosen instance *inst* of *ADDER*. There is an event *Comp* for every instance of *ADDER* in use. The notifications *InitNotif* and *DoneNotif* are not translated as such into the B Method. Though, the invariant should explicitly say that the *guarantees*-predicates in one of the notifications holds when event *Comp* is not enabled.

5 Refinement in Distributed B

We introduce a new type of refinement machine in Distributed B to deal with remote procedure calls and notification handlers in Event B. The *grid refinement machines* refine Event B systems, grid service machines, as well as other grid refinement machines. In a refinement step in Distributed B variables and events can be refined in the same way as in Event B. The substitutions in the remote procedures and the notification handlers are also refined as the events to reflect the changes of the variables. Note that the variables of the abstract grid service machines are global variables and may not be refined.

A grid refinement machine contains a new structuring mechanism in B that enables the grid refinement to obtain instances of the grid service machines via the index service. When the grid refinement machine has obtained a grid service instance, it can perform a remote procedure call to this instance and then wait for a notification from it.

5.1 Grammar for Refinements of grid services

The grammar of a grid refinement machine is an extension of the grammar of the refinement machine in Event B. For brevity we concentrate on the differences from the refinement machine.

```
Ref_gridservice ::= "GRID_REFINEMENT" Name
                    "REFINES" Name
                    Clause_ref_gridservice+
Clause_ref_gridservice ::=
   Clause_refinement |
   Clause_references |
   Clause_rpcs |
   Clause_notif_handlers
```

```
Clause_references ::= "REFERENCES" Name+,
Clause_rpcs ::= "REMOTE_PROCEDURES" Rpc_oper+;
Rpc_oper ::= Header_operation "=" NG_Substitution
Clause_notif_handlers ::= "NOTIFICA-
TION_HANDLERS" Notif_handler+;
Notif_handler ::= Name "=" "NOTIFICATION" Name
                           "SOURCE" Name ":" Name
                           "THEN" NG_Substitution "END"
```

In the *references*-clause we give the names of the grid service machines that the grid refinement machine can access and obtain instances of. The refined remote procedures are given in the *remote_procedures*-clause. Notifications are handled by special events, *notification*-substitutions, in the *notification_handlers*-clause. There is one notification handler event for each notification in the referenced grid service machines. The source of the notification is given as *<instance>:<grid service machine>*. The notification handlers should be implementable and not contain guarded substitutions. The *notifications*-clause that we introduced for grid service machines is not included in the refinement, since the *guarantees*-predicate of a notification should not be refined.

5.2 Translation of the refinement to B

In order to be able to show that the grid refinement machine is a correct refinement of another machine, e.g., an Event B specification or a grid service machine, both the grid refinement machine and its referenced grid service machines need to be translated to the B Method. Note that when we refine a grid service machine, we actually refine the instances of the grid service. In the translation from Distributed B to the B Method the instances of grid refinement machines are treated in the same way as the grid service machines.

In figure 2 refinement machine $\mathcal{C}_1$ is refined by the composition $\mathcal{C}_2$ *references* $\mathcal{A}$. The grid refinement machine $\mathcal{C}_2$ is translated to the refinement machine $\mathcal{C}_2_V$ and the grid service machine $\mathcal{A}$ is translated to the abstract machine $\mathcal{A}_V$. The *references*-relation between $\mathcal{C}_2$ and $\mathcal{A}$ is translated to an *includes*-relation between $\mathcal{C}_2_V$ and $\mathcal{A}_V$.

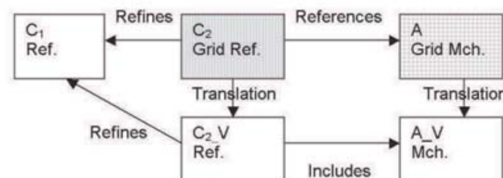

Figure 2: Translation to Event B

5.2.1 Managing instances of grid service machines

In the grid refinement machines we give instances of referenced grid service machines as ordinary variables. The instance *aa* of grid service machine $\mathcal{A}$ is declared as variable *aa* of type A, $aa \in A$. This type declaration is translated to the predicate $aa \in A_Instances \cup \{A_null\}$ in the B Method.

The grid refinement machines refer to the variables and remote procedures of the instances of a grid service machine with the notation *<instance>.<variable>* and *<instance>.<procedure>*, respectively. The variables of the grid service machine can be referred to only in the invariant of the grid refinement machine.

The remote procedure calls need to be translated to match the corresponding procedure definitions of the translated grid service machine. A call to a remote procedure $Proc(p)$ in instance *aa*, $aa.Proc(p)$ is translated to procedure call $Proc(aa,\ p)$ in the B Method, where the instance *aa* is given as an additional parameter.

5.2.2 Notifications

Notifications in a grid service machine inform the client that all the events in the grid service machine instance has become disabled. A notification handler in the client ensures that proper actions are taken for each notification. In the grid refinement machine a notification handler is expressed with the *notification_handler*-substitution:

$$\begin{array}{l} Handler \mathrel{\hat{=}} \\ \textbf{NOTIFICATION}\ Notif \\ \textbf{SOURCE}\ inst \in A \\ \textbf{THEN}\ T(inst) \\ \textbf{END} \end{array}$$

Here *Notif* is the notification to be handled, the source $inst \in A$ stands for the instance *inst* of the grid service $\mathcal{A}$ that sent the notification, and $T(inst)$ is a non-guarded substitution that refers to instance *inst*. Note that T can only make read-only remote procedure calls to instances of $\mathcal{A}$. A notification handler in a client is only enabled when all the events in the corresponding grid service have become disabled and the *guarantees*-predicate of the corresponding notification holds. Since a notification handler should only be executed once for each notification, it must disable itself.

Let us assume that we have a grid refinement $\mathcal{C}_2$ that refines an Event B specification $\mathcal{C}_1$ and that $\mathcal{C}_2$ has a reference to grid service machine $\mathcal{A}$, as shown in Figure 2. Furthermore, let $\mathcal{C}_2$ be the composition of the event systems $\mathcal{C}_{2h}$ containing the notification handlers (*notification_handler*-substitutions) and $\mathcal{C}_{2e}$ containing the rest of the events in grid refinement machine $\mathcal{C}_2$, $\mathcal{C}_2 = \mathcal{C}_{2h} \parallel \mathcal{C}_{2e}$. If we denote the composition of grid refinement $\mathcal{C}_2$ and its referenced grid service $\mathcal{A}$ with $\mathcal{C}'_2$, we have that event system $\mathcal{C}'_2$ is defined as:

$$\mathcal{C}'_2 \mathrel{\hat{=}} (\mathcal{A}//\mathcal{C}_{2h})\|\mathcal{C}_{2e}$$

where the events in the grid service machine $\mathcal{A}$ have a higher priority than the notification handlers in $\mathcal{C}_2$ and, hence, the notification handlers are executed only after the events in $\mathcal{A}$ have become disabled. In order to ensure the correct behaviour of the notification handling, the following conditions should hold:

$$\mathrm{gd}(\mathcal{C}_{2h}) \;\Rightarrow\; \neg\mathrm{gd}(\mathcal{A}) \tag{1}$$

$$\mathrm{gd}(C_{2h}) \Rightarrow \mathrm{wp}(C_{2h}, \neg\mathrm{gd}(C_{2h})) \tag{2}$$

Condition (1) is derived from the prioritised composition $\mathcal{C}_2'$. It states that all events in grid service machine $\mathcal{A}$ are disabled when a notification handling event is enabled. The notification handler C_{2h} in event system $\mathcal{C}_{2h}$ can only be executed once for each notification it receives and, hence, it must disable itself as stated in condition (2).

The event system $\mathcal{C}_{2e}$ can further be considered to be the parallel composition of an event system $\mathcal{C}_{2rpc}$ containing the events making remote procedure calls and a system $\mathcal{C}_{2o}$ containing the rest of the events of $\mathcal{C}_{2e}$, $\mathcal{C}_{2e} = \mathcal{C}_{2rpc} \parallel \mathcal{C}_{2o}$. An event C_{2o} in system $\mathcal{C}_{2o}$ should not interfere with the notification handlers in $\mathcal{C}_{2h}$ by enabling or disabling them as stated by conditions (3) and (4).

$$\mathrm{gd}(C_{2o}) \wedge \mathrm{gd}(\mathcal{C}_{2h}) \Rightarrow \mathrm{wp}(C_{2o}, \mathrm{gd}(\mathcal{C}_{2h})) \tag{3}$$

$$\mathrm{gd}(C_{2o}) \wedge \neg\mathrm{gd}(\mathcal{C}_{2h}) \Rightarrow \mathrm{wp}(C_{2o}, \neg\mathrm{gd}(\mathcal{C}_{2h})) \tag{4}$$

The conditions (1) - (4) above are fulfilled by introducing extra features upon translating the grid refinement $\mathcal{C}_2$ to the B Method. Firstly, we introduce a boolean variable *A_notification* for each referenced grid service machine $\mathcal{A}$:

$$A_notification \in A_Instances \rightarrow BOOL$$

When the variable $A_notification(inst)$ has the value *true,* the grid refinement $\mathcal{C}_2$ is prepared to receive a notification from instance *inst* of $\mathcal{A}$. The notification handler *Handler* is translated to take variable *A_notification* into consideration:

$Handler \,\hat{=}$
ANY $inst$ **WHERE**
  $inst \in A_Instances \wedge$
  $\neg\mathrm{gd}(\mathcal{A}(inst)) \wedge Q_{Notif}(inst) \wedge$
  $A_notification(inst) = TRUE$
THEN $T(inst) \parallel A_notification(inst) := FALSE$
END

The guard of the translated notification handler *Handler* states that the events of the grid service machine $\mathcal{A}$ for the instance *inst* should be disabled when *Handler* is enabled, $\neg\mathrm{gd}(\mathcal{A}(inst))$, ensuring that condition (1) is fulfilled. Predicate Q_{Notif} is obtained from the *guarantees*-statement of the corresponding notification *Notif* in $\mathcal{A}$ and models the condition for this notification to be sent. The

condition $A_notification(inst) = TRUE$ in the guard of the translated notification handler states that the grid refinement is prepared to receive a notification. In order to ensure condition (2) stating that a notification handler is executed only once for each notification, the assignment $A_notification(inst) :=$ $FALSE$ is added to *Handler* upon translation. In each event of $\mathcal{C}_2$ the assignment $A_notification(inst) := TRUE$ is added after the remote procedure calls to procedures in instance *inst* of $\mathcal{A}$ to prepare the notification handlers to receive a notification. Note that this assignment is also added after a call to A_GetNew for a new instance *inst* of $\mathcal{A}$.

The guards of the notification handlers, gd$(\mathcal{C}_{2h})$, refer to the variables of $\mathcal{A}$, as well as the variable *A_notification*. Since, an event C_{2o} in the event system $\mathcal{C}_{2o}$ does not modify these variables, conditions (3) and (4) hold trivially.

5.2.3 Example of a grid refinement machine

Let us give an example of a grid refinement machine and its translation to B. The grid refinement machine *CLIENT1* below sums up a number of sub-sums (here 100), $(\sum_{counter=1}^{100} \sum_{j=0}^{counter} j)$. The sub-sums from 0 to *counter* are computed in the grid service machine *ADDER* presented in the example in Subsection 4.2.2. The current instance of grid service machine *ADDER* used for the sum computation is given by variable *adder*. The variable *counter* keeps track of the number of calls made to instance *adder*, while variable *total* gives the current result of the sum computation. The variable *rpc* states whether there is a computation in progress in *adder* or not. Event *Evt* of *CLIENT1* initiates the computation by a call to procedure *SetNewData* in instance *adder*. *CLIENT1* then waits for a notification to update variable *total* with the sub-sum computed by *adder*.

The grid refinement machine *CLIENT1* in Distributed B is translated to the refinement machine *CLIENT1_VERIFICATION* in the B Method as follows:

GRID_REFINEMENT	**REFINEMENT**
$CLIENT1$	$CLIENT1_VERIFICATION$
REFINES	**REFINES**
$CLIENT$	$CLIENT$
REFERENCES	**INCLUDES**
$ADDER$	$ADDER_VERIFICATION$
	PROMOTES
	$Comp$
VARIABLES	**VARIABLES**
$counter, total, rpc, adder$	$counter, total, rpc, adder$
INVARIANT	**INVARIANT**
$counter \in \mathbb{N} \wedge total \in \mathbb{N} \wedge$	$counter \in \mathbb{N} \wedge total \in \mathbb{N} \wedge$
$rpc \in BOOL \wedge$	$rpc \in BOOL \wedge$
$adder \in ADDER$	$adder \in ADDER_Instances \cup$
	$\{ADDER_null\}$
$\ldots$	$\ldots$

```
cont.

INITIALISATION
  counter := 0; total := 0;
  rpc := FALSE;
  adder :∈ ADDER

EVENTS
...
Evt ≙
  SELECT rpc = FALSE ∧ counter < 100
  THEN
    counter := counter + 1; rpc := TRUE;
    adder.SetNewValue(counter)
  END

NOTIFICATION_HANDLERS
Handler ≙
  NOTIFICATION DoneNotif
  SOURCE inst ∈ ADDER
  THEN
    VAR val IN
     val ← inst.GetResult;
     total := total + val;
     rpc := FALSE
    END
  END
END
```

```
cont.

INITIALISATION
  counter := 0; total := 0;
  rpc := FALSE;
  adder :∈ ADDER_Instances;
  ADDER_notification(adder) := TRUE

OPERATIONS
...
Evt ≙
  SELECT rpc = FALSE ∧ counter < 100
  THEN
    counter := counter + 1; rpc := TRUE;
    SetNewValue(adder, counter);
    ADDER_notification(adder) := TRUE
  END ;
Handler ≙
  ANY inst WHERE
    inst ∈ ADDER_Instances∧
    ¬(state(inst) = start)∧
    state(inst) = done∧
    ADDER_notification(inst) = TRUE
  THEN
    VAR val IN
     val ← GetResult(inst);
     total := total + val;
     rpc := FALSE
    END ;
    ADDER_notification(inst) := FALSE
  END
END
```

Upon translation variable *adder* is transformed into an instance type of the grid service machine *ADDER*, $adder \in ADDER_Instances \cup \{ADDER_null\}$. In the remote procedure call $SetNewValue$ the instance $adder$ is introduced as a parameter, $SetNewValue(adder,\ counter)$. The notification handler *Handler* is translated to a new notification handling event for every instance $inst$ of $ADDER$. The variable *ADDER_notification* is taken into consideration in the notification handler, as well as after remote procedure calls in the events, in order to ensure that notification handler is executed once for each notification.

5.3 Proofs

In order to show that the grid refinement is a correct refinement of a more abstract system the proof obligations given in Subsection 2.3 need to be generated and discharged. The proof obligations concerning the refinement of the initialisation, the procedures, as well as the events are generated automatically by the tools of the B Method. Furthermore, the proof obligation for showing that the refined system does not terminate more often than the abstract system can also be directly generated by these tools (via Event B). In order to show that the new events terminate when executed in isolation, a variant that is decreased upon execution of each new event is needed in the grid refinement machine. Note that the notification handlers are introduced as new events. For the notification handlers dealing with notifica-

tions from grid service machine $\mathcal{A}$ the variant is the number of instances for which the notification has not yet been sent:

$$\mathrm{card}(\{inst | inst \in A_Instances \wedge A_notification(inst) = TRUE\})$$

This variant assumes that new events do not call procedures in $\mathcal{A}$. The proof obligation ensuring that a refined remote procedure is enabled when the corresponding abstract remote procedure is enabled is *true* by construction, since the remote procedures contain only non-guarded substitutions. Hence, all the proof obligations for proving the correctness of a refinement step in Distributed B can be automatically generated with the tool support for the B Method (via Event B). These proof obligations can then be automatically or interactively discharged with the help of these tools.

6 Implementation

The grid system development in 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 translated to Java. The remote procedures and notification handlers are constructed in such a way that they can be directly translated [19]. Furthermore, all the variables except for instances of grid service machines can be directly translated [19]. The instances are translated to objects encapsulating the grid specific features. The handling of grid service instances is performed via the API's for grid services and for service data provided by Globus Toolkit. This grid specific code can be inserted into the initialisation code for the grid services and in the procedures *GetNew* and *Destroy*. In order to be able to implement an event system, all the events in the *events*-clause need to be merged [2] into one single event. The composed event can be translated to a *while*-loop in Java as follows:

SELECT $G1$ **THEN** $S1$
...
WHEN Gn **THEN** Sn
END

```
while (true) {
   if(G1) S1;
   ...
   else if(Gn) Sn;
   else break;
}
```

The *break*-statement terminates the loop when the event is disabled. Note that the event can only be implemented, if all the guards *Gi* and the substitutions *Si* belong to the B0 language.

The sending of notifications should be taken into account when translating a grid refinement of a grid service machine (grid service) to Java. The event system is translated to an infinite loop where a notification is sent when the event is not enabled.

```
while(true){
   synchronized(this){
      if(G1) S1;
      ...
      else if(Gn) Sn;
      else{
         if(Qnotif)
            sendNotif();
         ...;
         wait();
      }
   }
}
```

The *break*-statement in the translation of the event system to Java is replaced by statements for sending notifications. Since at least one condition Q in a *guarantees*-clause holds when the event is disabled, there will always be a notification to send. After the notification has been sent the system waits until it is notified of a remote procedure call, in order to be able to continue the execution. The sending of a notification with condition $Qnotif$ in the *guarantees*-clause is encapsulated in the method $sendNotif$. The statement $synchronized(this)$ ensures that events of this event system are atomic.

In the Java translation of a grid refinement of an Event B specification (a client) the handling of notifications is considered. The notification handlers for the instances created by the procedure *GetNew* are registered in the Globus Toolkit middleware. The notification handler is then automatically executed for the appropriate instance every time a notification is received.

In Distributed B the sending and handling of notifications are performed as one atomic event. In order to achieve the same behaviour in Java a sequence number variable, *seqNum*, for each instance in both the client and grid service is included upon translation. The sequence number is needed, since in Java a remote procedure can be called in a grid service after a notification has been sent from the grid service to the client, but before it has been handled. This means that the notification handler would be executed when the condition Q in the *guarantees*-clause in the grid service does not hold. The following algorithm is used in order to take notice of valid notifications only.

1. The sequence number of both the grid service and the client are initialised to 0

2. When the client calls a remote procedure, it increments the sequence number and send it to the grid service. The grid service updates its sequence number to this value.

3. The grid service sends its sequence number with the notification. The client checks if the received sequence number of the notification is the same as its

current sequence number for that grid service. If the numbers are the same, the notification handler is executed. In case the received sequence number is less than the current sequence number of the client, the grid service has not completed all its tasks and the notification is discarded.

4. The sequence number is reset to 0 when the notification handler has been executed.

A new sequence number *totSeqNum,* denoting the sum of all variables *seqNum,* is introduced for detecting termination in the presence of notifications. It is increased when remote procedures are called and decreased when a notification handler is executed. The event system of a grid refinement modelling a client is not allowed to terminate when there are pending notifications to be handled. Hence, the system only terminates if the event is disabled and the sequence number *totSeqNum* is equal to zero.

```
while(true){
   synchronized(this){
      if(G1) S1;
      ...
      else if(Gn) Sn;
      else{
         if(totSeqNum>0)
            ...
         else
            break;
      }
   }
}
```

The translated event system contains an *if*-statement that checks if notifications are pending, $totSeqNum > 0$. If there are no notifications pending the event system terminates.

After we have translated the Distributed B code to Java and all the grid specific features have been handled, we have implemented the grid system in a formal manner where the implementation is proved correct with respect to its specification.

7 Conclusions

In this paper we have proposed a language Distributed B that extends Event B for designing and implementing correct grid systems. Grid systems are large distributed systems and standard development tools cannot guarantee their correct implementation. We introduced two new types of machines, *grid service machine*

and *grid refinement machine,* for handling grid specific issues in Event B. We proposed a method where the development of a grid system starts with refinement within Event B. After a number of refinement steps the system is split up into a grid refinement machine and a number of grid service machines in Distributed B. These machines can then be further refined. Throughout the development in Distributed B the grid constructs are translated to machines in the B Method for verification purposes. The machines are introduced in a manner that allows automatic generation of the necessary proof obligations. Furthermore, the concrete specifications can be automatically translated to executable code, since the grid constructs have been introduced in such a way that they ensure that the system will be implementable. Hence, we have introduced a method for implementing grid systems where the implementation can be proved correct with respect to its specification.

The B language has earlier been successfully used for modelling distributed systems, e.g., in [20]. These examples do, however, not consider implementation issues of the developed specification. Implementation of distributed systems using the B Method has also been considered for the combination of ordinary B and CORBA in [16]. Though, the paper does not consider concurrent behaviour and dynamic management of instances of distributed components. Other formal methods have also been extended previously to enable implementation of distributed systems using different application domains. For example, the DisCo formalism has been used for designing and implementing systems that were translated to Enterprise Java Beans (EJB) [15]. Grid specific features were not considered in that extension.

The architecture of the systems developed with Distributed B forms a tree of grid services. Even if this is a very common architecture for grid systems, it might be too restrictive in some cases. Hence, we plan to investigate also other architectures. In the modelling of grid systems in distributed B we have made the assumption that no network failures occur. In future versions of Distributed B also network failures and node failures will be taken into consideration. Moreover, we consider development of tool support for grid systems in Distributed B.

The language Distributed B that we proposed in this paper can provide a convenient formal development process for grid systems. The systems will by construction have an architecture that is implementable. Furthermore, specifications of grid systems constructed in this language will be clear to understand, since the systems are modeled in terms of grid primitives with a precise meaning. We believe that our approach to adapt Event B to the Globus Toolkit middleware can also be useful for other types of middleware for distributed systems.

References

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

[2] J. R. Abrial. Event Driven Sequential Program Construction, 2001. http://www.atelierb.societe.com/ressources/articles/seq.pdf. (accessed 28.10.2004)

[3] J. R. Abrial and L. Mussat. Event B Reference Manual, 2001. http://www.atelierb.societe.com/ressources/evt2b/eventb_reference_manual.pdf. (accessed 28.10.2004)

[4] N. Aguirre, J. Bicarregui, T. Dimitrakos and T. Maibaum. Towards Dynamic Population Management of Abstract Machines in the B Method. In D. Bert, editor, *Proceedings of the Third international conference of B and Z users: ZB2003*. LNCS 2651. Turku, Finland, pp. 528-545. Springer-Verlag, 2003.

[5] 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.

[6] R. J. R. Back and K. Sere. From modular systems to action systems. In *Software - Concepts and Tools*, 17:26-39, 1996.

[7] M. Butler and M. Waldén. Parallel programming with the B Method. Chapter 5 in E. Sekerinski and K. Sere. (eds.) *Program Development by Refinement - Case Studies Using the B Method*, pp. 183-195. Springer-Verlag, 1998.

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

[9] 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.

[10] I. Foster, C. Kesselman, J. Nick and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Argonne National Laboratory, 2002. http://www.globus.org/research/papers/ogsa.pdf. (accessed 28.10.2004)

[11] Globus Toolkit. The Globus Alliance, 2004. http://www.globus.org/. (accessed 28.10.2004)

[12] 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 28.10.2004)

[13] E. J. Hedman, J. N. Kok and K. Sere. Coordinating Action Systems. *Theoretical Computer Science,* 240:91-115. Elsevier Science, 2000.

[14] G. Mair and A. Villazón. Implementing a Distributed Master/Slave Grid Service with Globus Toolkit 3 (GT3). `http://dps.uibk.ac.at/~gregor/mandel.pdf`, 2003. (accessed 28.10.2004)

[15] R. Pitkänen. A Specification-Driven Approach to Development of Enterprise Systems. In *Proceedings of NWPER'2004 - 11th Nordic Workshop on Programming and Software Development Tools and Techniques*, TUCS General Publication 34. Turku, Finland, 2004.

[16] O. Rolland and T. Muntean. Refining Open Distributed Systems to CORBA. In *Proceedings of RCS'02- International workshop on refinement of critical systems: methods, tools and experience*. Grenoble, France, 2002.

[17] C. Snook and M. Waldén. Use of U2B for specifying B action systems. In *Proceedings of RCS'02- International workshop on refinement of critical systems: methods, tools and experience*. Grenoble, France, 2002.

[18] K. Sere and M. Waldén. Data Refinement of Remote Procedures. *Formal Aspects of Computing*, 12(4):278-297, 2000.

[19] 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

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

TURKU
CENTRE *for*
COMPUTER
SCIENCE

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

University of Turku
- Department of Information Technology
- Department of Mathematical Sciences

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 952-12-1445-5
ISSN 1239-1891