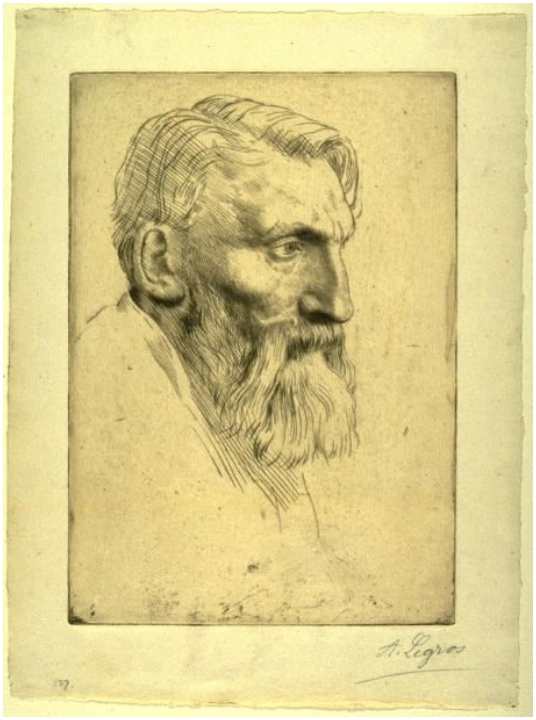
Project IST-511599

RODIN

"Rigorous Open Development Environment for Complex Systems"



RODIN Deliverable D19

Intermediate report on methodology

Editor: C. B. Jones (Newcastle University)

Public Document

29th August 2006

http://rodin.cs.ncl.ac.uk/

Abstract

One aim of the Rodin project is to contribute *formal methods* which will underpin the creation of *fault-tolerant systems*. This intermediate report from WP2 (Methodology) describes progress during the second year of the Rodin project; it also discusses our plans for the final deliverable on methodology.

Contributors:

Many people have written material for Chapters 3 and 2; specific contributions include:

Section 2.1 written by Linas Laibinis
Section 2.2 written by Ian Johnson
Section 2.3 written by Ian Oliver
Section 2.4 written by Neil Evans and Michael Butler (on behalf of Praxis)
Section 2.5 written by Maciej Koutny
Section 3.1 written by Maciej Koutny
Section 3.2 written by A. Iliasov, A. Romanovsky, E. Troubitsyna, L. Laibinis
Section 3.3 written by Joey Coleman and Cliff Jones
Section 3.4 written by Manoranjan Satpathy, Qaisar A. Malik and Johan Lilius
Section 3.5 written by Elena Troubitsyna
Section 3.6 written by Michael Butler
Section 3.7 written by Joey Coleman and Cliff Jones
Section 3.8 written by Divakar Yadav and Michael Butler
Section 3.9 written by Fernando Castor Filho, Alexander Romanovsky, and Cecília Mary F. Rubira
Section 3.10 written by Alexei Iliasov, Victor Khomenko, Maciej Koutny, Apostolos Niaouris and Alexander Romanovsky

Contents

Chapter 1

Introduction

This chapter indicates progress on technical issues which were identified in D9 (the "Preliminary report on Methodology"). Overall, we feel that very good progress is being made on issues of development methods. Chapter 2 sets out methodological insights which have come from the case studies (this can of course be read in conjunction with D18). The variety of the case studies is having the desired effect of stretching our ideas on methods. Chapter 3 –like D9– provides a series of "essays" on methodological issues. These vary considerably in length — to a large extent this is caused by whether publications exist to which the essay can point for technical details.[1]

Possible plans for the final deliverable on methodology are set out in Chapter 4 (and one structure for the final document sketched in Appendix A). We should be particularly grateful for input at the October 2006 Project Review on the alternatives for the final methodology document(s).

Figure 1.1 provides a reminder of the list "issues" identified in D9: they are provided with "I-n" numbers for reference elsewhere in this (D19) report. Comments on progress on these issues:

I-1 Building a specification from parts We see this as an important issue for "scaling"; it is addressed under §2.4 below.

I-2 Partial functions This is not seen as a significant issue — we shall return to it during proof experiments with the new Rodin prover but the current Event B position is certainly adequate. (A further paper has been produced [Jon06b].)

I-3 Role of invariants This is not seen as a significant issue — we'll return to it during proof experiments with the new Rodin prover.

I-4 Controlling the order of operation execution Work is on-going but there have been no significant publications on this topic (but see [Jon05b]).

[1] It is also worth mentioning that Section 2.4 is rather longer than the other case study sections because a number of technical issues are included in the text.

Number	D9 Section	Brief description
I-1	§2.1.1	Building a specification from parts
I-2	§2.1.2	Partial functions
I-3	§2.1.3	Role of invariants
I-4	§2.1.4	Controlling the order of operation execution
I-5	§2.2.1	Deriving specifications (of “control systems”)
I-6	§2.2.2	Domain modelling
I-7	§2.3.1	Comping with interference
I-8	§2.3.2	Refining atomicity
I-9	§2.3.3	Process algebras and net theory
I-10	§2.3.4	Process algebra and Event B
I-11	§2.4.1	Failure management
I-12	§2.4.2	Determining the failure specification of a system
I-13	§2.4.3	BPEL-like languages
I-14	§2.5.1	Synergy between model checking and reasoning
I-15	§2.5.2	Rigorous reasoning
I-16	§2.5.3	Role of programming languages
I-17	§3	Requirements structure
I-18	§4	Linking UML and B
I-19	§5	Records in Event B
I-20	§6	Methodology for mobile systems
I-21	§7	Model driven development
I-22	§8	Exception handling in mobile environments

Figure 1.1: Reference numbers from issues in D19

I-5 Deriving specifications (of "control systems") This has been an active area; an update is given in §3.3 below.

I-6 Domain modelling There has been relevant activity in several areas — here, see in particular Section 2.3. Before the final report is written, we will position each of the case studies with respect to domains.

I-7 Coping with interference Addressed under §3.7 below.

I-8 Refining atomicity See §2.3, §3.2, §3.6, §3.8, §3.1, §3.11 below.

I-9 Process algebras and net theory This too is an active area; here, it is touched on under §3.1 below.

I-10 Process algebra and Event B Another active area; it is touched on under §3.1 below.

I-11 Failure management Addressed under §2.2 below.

I-12 Determining the failure specification of a system Addressed under §3.5 below.

I-13 BPEL-like languages No further work is envisaged on this topic.

I-14 Synergy between model checking and reasoning We have thought about and discussed this area but await experiments with the main Rodin tools and Plugins to draw conclusions (e.g. the encouraging performance results with model checking pi-calculus will certainly affect how a user will experiment with model-checking before attempting proofs).

I-15 Rigorous reasoning We will be experimenting on this with the new proof tool.

I-16 Role of programming languages Jean-Raymond Abrial talked about this in his VSTTE position statement. Also addressed under §2.3 below.

I-17 Requirements structure We are using the ideas set out in D9 and will report more in further deliverables.

I-18 Linking UML and B Addressed under §2.3 below.

I-19 Records in Event B Addressed under §2.4 below.

I-20 Methodology for mobile systems Addressed under §2.5, §3.2, §3.10, §3.11 below.

I-21 Model driven development Addressed under §2.1, §2.3 below.

I-22 Exception handling in mobile environments Addressed under §2.1, §2.5, §3.9, §3.2, §3.11 below.

Generic specifications Addressed under §2.4 below.

Testing Addressed under §3.4 below.

Transience Addressed under §3.5 (and indirectly –see references– in§3.3) below.

Special needs of users unfamiliar with formal methods Addressed under §2.1, §2.3 below.

Integration with less formal methods Addressed under §2.3, §3.11 below and in our ongoing "UML/B" work.

Link to Rodin Tasks

The Rodin Description of work defines the following tasks, the link with the sections here is as follows

Task	Description	§n
T2.1	Formal representations of architectural design, decomposition and mapping principles	§2.1 §2.2 §2.3 §2.4 §2.5 §3.7 §3.2 §3.11
T2.2	Reusability, genericity, refinement	§2.4 §2.3 §3.4 §3.2 §3.6 §3.8 §3.1 §2.1 §2.3 §3.11
T2.3	Development templates for fault-tolerant design methods	§2.2 §3.3 §3.5 §3.11 §3.2
T2.4	Development templates for reconfigurability, adaptability and mobility	§2.4 §2.5 §3.2 §3.11
T2.5	Requirements evolution and traceability	§2.2

Chapter 2

What we are learning from the tension with the case studies

It was always our intention that the case studies would provide feedback to the evolving development methods. This is indeed proving to be the case. Here, in this intermediate deliverable from WP2, we specifically identify some of the things we are learning.

2.1 Case Study 1: Formal Approaches in Protocol Engineering

The work on Case Study 1 –Formal Approaches in Protocol Engineering– focuses on formalisation and verification of the design method Lyra. Lyra is an UML2-based service-oriented method for development of telecommunication systems and communication protocols. In the first year of the RODIN project we have developed formal specification and refinement patterns reflecting essential Lyra models and transformations. This allowed us to verify the Lyra development steps (phases) on the basis of stepwise refinement of a formal system model in the B Method. This work has been reported in [LTL+05].

During the second year our work has progressed in two directions. First, the further development of the specification and refinement patterns to incorporate the fault tolerance mechanisms used in the domain [LTL+06]. Second, the development of an approach to formal verification of consistency of the UML2-based Lyra development [LIM+05]. Let us now describe these methodological advances in more detail.

To incorporate formal reasoning about fault tolerance into the formalized Lyra development flow, the specification and refinement patterns for Lyra models have been extended with explicit representation of possible errors and error recovery. The extension has affected the specifications of service components directly responsible for controlling the service execution flow (called service directors). The recovery mechanisms allowing a service director to retry the failed service execution as well as to ”roll back” in the service execution flow have been incorporated in the specification pattern of a service director. Moreover, in refinement steps modelling service decomposition and distribution over a given network, the fault tolerance mechanisms have been accordingly distributed over the involved service components. Termination of potentially infinite recovery process is guaranteed by modelling the maximal execution time that is gradually decreased by service execution.

To automate translation and verification of Lyra UML models in the B Method, we have developed an approach to verifying the consistency of the provided UML models. The approach consists of formalisation of the intra-consistency (i.e., expressing the relationships between models within the same Lyra development phase) and inter-consistency (i.e., the relationships between different Lyra phases) rules for the Lyra UML models. The formalisation is done using the B Method in such a way that the requirements are gradually (i.e., phase by phase) introduced and incorporated by the corresponding B refinement steps. The achieved results create a basis for developing a formally verified UML profile for Lyra.

2.2 Case Study 2: Engine Failure Management System

Rodin methods and technology such as UML-B have shown promise in tackling Failure Management Domain concerns for ATEC such as closing the semantic gap (i.e., closer mapping of the problem domain to the design) and providing a reliable reusable process to meet the demands of a safety environment.

In the first year the University of Southampton in cooperation with ATEC developed a generic model of the failure management system (FMS) based on a UML-B profile. The 2nd year work on the case material consists of methodological contributions by ATEC, University of Southampton (Soton) and Aabo Akademi (Aabo). Which have been outlined in the RODIN D18 deliverable.

ATEC have been evaluating emerging RODIN methodology by undertaking a pilot study of the case material. Their work has demonstrated the usability of the methodology and its tools by a novice user of formal methods and B. However, it was felt more guidance in model development should be developed and this is to be addressed in collaborative research with Aabo Akademi and the University of Southampton. ATEC's exploration of methodology has also identified a need to provide more flexibility in the refinement process in order to identify valid requirements early from which a more rigorous refinement chain can be constructed. A process to address this issue was investigated which has been called Idealisation-De-idealisation. Idealisation supports the idea that obtaining good abstractions are difficult initially and that less rigorous development can initially be undertaken to establish the main functionality of a model. De-idealisation refers to developing the model rigorously to obtain consistency in the refinement chain..

The University of Southampton has continued developing methodology supporting the generic model. Work at Southampton has proposed a prototype process for the Verification and Validation of a generic specification of this type, demonstrating stage (1): validate structural model using test data — and stage (2): verify system instance data against structural instance model. This was done using the existing UML-B tool and ProB model checker. The specification has been decomposed into features as the first step in an investigation of feature-based description, refinement and composition of generic specifications. This investigation will establish how to structure such feature-based transformations using the relevant mechanisms of the Event-B language: refinement, decomposition and generic instantiation. A student project group at the University of Southampton has developed a plugin for UML-B, the Requirements Manager. This tool is a PostgreSQL-based repository of FM system instance data, with functions to input and verify instance data against the generic model, and to upload the data to UML-B for generation of a system instance UML-B specification. As a user-acceptance test of

the tool, the Verification and Validation exercise of the previous paragraph has been performed with a full system instance dataset.

Aabo Akademi has been working with classical refinement development of the FMS (see Section 3.5). The main result of developing the FMS by stepwise refinement in B is a set of formal templates for specifying and refining the FMS. The developed FMS is able to cope with transient faults occurring in the system with multiple homogeneous analogue sensors. The formal templates specify sensor recovery after the occurrence of transient faults and ensure the non-propagation of errors further into the system.

2.3 Case Study 3: MDA

First to note is that overall Nokia is not a general user of 'formal methods' as, say, some automotive or aerospace companies might be. However it is well known that we have a need for analysis of the designs that we are producing.

Secondly is that we already have well defined (and in some cases ingrained) processes and methods for the development of the various kinds of embedded and real-time systems that are found in mobile devices.

It is often the case that the adoption of new development technologies *requires* a wholesale change in the way systems are developed. It is then often also the case that the new method fails because of the burden of trying adapt it and the existing processes to each other.

We take the view that new techniques such as those being developed inside Rodin should be agnostic to the underlying processes and be compatible with existing techniques and methods as far as possible. This is to minimise the disruption caused when introducing these techniques.

2.3.1 Development

The use of formal methods is already in place, albeit in a minor way due to the plethora of notations and in most cases the seeming lack of integration with the defacto languages and processes — particularly the UML.

The first year of Rodin concentrated more on the use of B for the specification of the NoTA (Network on Terminal Architecure) system. Here we had the chance to compare the use of UML+B and plain B with already existing models and an implementation constructed through the use of "agile methods" and the SDL language.

The results of the first year show that overall productivity increases and the amount of design errors significantly decreases when using a formal method. The caveat here is that the use of formal methods must be 'pragmatic' rather than 'dogmatic' in nature. The first versions of the upper layers of the NoTA protocol stacks based upon the verified designs are currently being implemented and tested in production environments.

The second year has focussed much more on the superstructure that Rodin must fit into in order to gain acceptance and be used with our model based methods. Later sections here discuss what we mean by MDA and MDE in this context.

Work with Abo Akademi has resulted in two threads of development: the first concentrating on patterns expressed through 'MDA-style transformations' for adding fault tolerance aspects to a B specification; and secondly more formal underpinnings for 'model based' structures.

Further work whcih is due to start during the third year of the project will focus on hardware specification language generation from EventB specifications.

Some work has been proposed with University of Southampton on additions to U2B — some of which have been prototyped locally and product line structures.

(The work between ATEC and University of Southampton is being followed closely.)

2.3.2 Model Driven 'XXX'

The term 'model driven" is becomming overused such that it can be almost applied as a prefix to any computing term: model, engineering, testing, development etc. However the (modern) origins of the term come from the OMG's Model Driven Architecture concept which attempts to provide a framework (Figure 2.1) for relating models of varying levels of abstraction together. The OMG's MDA also seeks to enforce the split between models of the system (domain, design, implemention) and the partitioning and mapping which are often called architecture models.

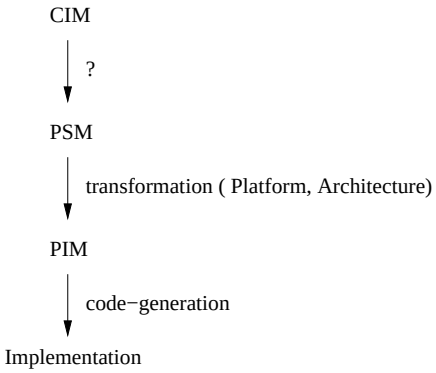


Figure 2.1: Model Driven Architecture Superstructure

The MDA is based around the ideas of *the* platform independent model (PIM) and the *the* platform specific model (PSM) bound together with a transformation which effectively encodes the architectural decisions made to move from the PIM to the PSM. There is also the notion of *the* computation independent model (CIM) which is transformed in a similar manner to a PIM; the semantics of the CIM is not well defined and is relegated to a single sentence in the official description of the MDA.

We take a broader view such that we have a collection of models which are related by a number of relationships. These relationships can be classified into various types depending on what aspects of the model they preserve. The typical example is that (as commonly described in MDA) of the transformation which is normally used to concretise one model into another on some given platform: the source of this tranformation being called the PIM the target(s) the PSM(s). Another example is that of the translation which maps a model in one language to an equivalent structure in another language, for example UML to B or CSP to TTCN/3.

2.3.3 MDA in Rodin

The OMG promotes the use of various flavours of the UML as the primary language(s) of models, while in Rodin we are more flexible in that we use the most suitable language whether

or not a translation to or from a flavour of UML is available.

We also work more in the sprit of ‘model driven’ in that we seek to create a superstructure in which models exist and can be related to each other. In our current NoTA work this superstructure appears in Figure 2.2.

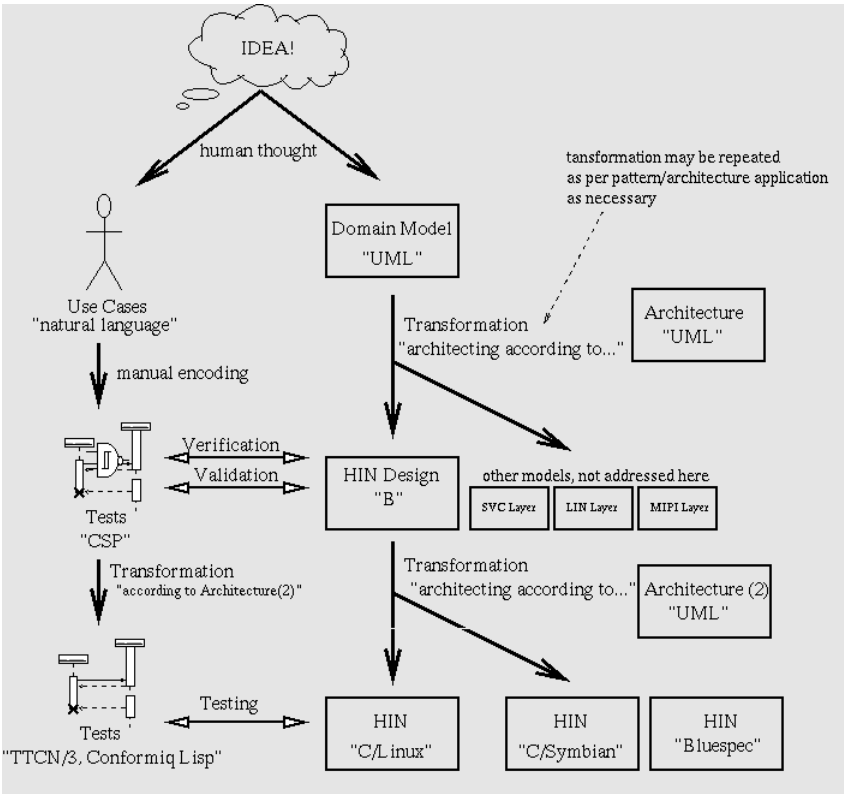


Figure 2.2: Model Superstructure

For simplicity we have shown only four levels in this diagram. The first level is always the idea which we intially encode as natural language use cases and a domain model developed using object oriented technqiues (and thus described using UML). For this first part a variety of methods or techniques can be used: we particularly favour CRC cards, Catalysis and Responsibility Based Design approaches.

The procedure of transforming between two models is the one of mapping a PIM to a PSM as described by the OMG where the choice of platform is made by application of a pattern (e.g. making a model MVC specific or applying a fault tolerance pattern) or in the more traditional sense (encoding a model into Java). This procedure may be repeated as many times as necessary.

Typically at high levels of abstraction one can map the model to a form where it can be verified. This in the case of Rodin is simply a mapping to B which is then verified by theorem proving.

At lower levels of abstraction and especially when the model has been partitioned more extensively (e.g. into HIN, LIN etc layers) it may be necessary to switch languages to something more suitable. This is often a cause of some concern for ‘model based’ affictionados where a

change between a graphical to a textual language is made. A model expressed in, for example, C is still a model, albeit one at a very platform specific (or independent) level.

Testing (Model Based of course) is facilitated by formalising the use cases as CSP expressions which can be 'run' against the model (i.e. the model is an oracle for those expressions). These expressions can be transformed in much the same manner as the models they are being run against to more suitable representations for the more concrete level. Typically one maps to more suitable testing languages such as TTCN/3 or Conformiq Lisp.

Transformation continues until the models are expressed in a form where execution by some suitable environment (e.g. x86 processor, compiler, DBMS) is possible.

While model driven approaches concentrate on the top to bottom concretising transformations and sometimes one or more translation-like mappings they do not concentrate on the properties that a transformation must adhere to. One aspect of Rodin is the emphasis on transformations that preserve refinement between a pair of models. Of course this is not always possible in all cases and properties of transformations need to be demonstrated via other route, for example via testing. Translations between languages are similar but must enforce that the models are isomorphic with respect to the information they convey.

One aspect not directly discussed here is that of relationships between model elements that exist on differing architectural partitionings. This normally suggests the usage of some kind of communication technology. This is should be handled by the transformation which should reify the relationship into a model in its own right which embodies the communication technology.

2.4 Case Study 4

2.4.1 Problems to be Overcome

There are two distinct drawbacks of the original approach to the CDIS development: first, the complexity of the system makes the formal specification necessarily complex and difficult to visualise, and second is the lack of continuity from the specification to the design. In the *idealised* specification, updates are performed instantaneously at all user positions whilst, in the actual system, there is an inevitable delay because the information must be distributed to the user positions over a network. Hence, there is no natural refinement of the specification (in the usual sense of the word) to any realistic design. We have been investigating more novel notions of specification and refinement (in Event-B) to make the specification more comprehensible and to find a suitable link between the specification and design viewpoints. In this section, we begin by outlining our solution to the first of these problems. In particular, we describe a *non-linear* refinement technique to introduce different aspects to the specification. Then we give an outline to a second *transformation* technique to move from the idealised view of the system to the design.

2.4.2 A Methodology for Specifying CDIS

As stated above, we begin with an idealised view of the system. We model a system that has a centralised database from which information can be retrieved. In order to get a better overview of the entire system, we follow a top-down approach. At the top level, we ignore

all of the airport-specific features to produce a specification describing a generic display system. Through an iterated refinement process, we introduce more features into the specification until all of the CDIS functionality is specified. At each step the tool generates a number of proof obligations which must be discharged to show that the refinements are consistent. Since each refinement introduces only a small part of the overall functionality, the number of proof obligations at each step is relatively small (approximately less than 20).

Generic Display Context

The purpose of CDIS is to enable the the storage, maintenance and display of data at user positions. If we ignore specific details about what is stored and displayed then CDIS becomes a 'generic' display system. We begin by constructing a specification for a generic system (which will be, of course, somewhat influenced by the original VDM specification) and, through subsequent refinements, introduce more and more airport-specific details so that we produce a specification of the necessary complexity, and reason about it along the way. By providing a top-down sequence of refinements it is possible to select an appropriate level of abstraction to view the system: an abstract overview can be obtained from higher level specifications whilst specific details can be obtained from lower levels.

Meta Data Context.

Rather than specifying individual airport attributes (such as wind speed) as state variables of a particular value type, two abstract types are introduced that correspond to the collection of attribute identifiers and attribute values. This allows us to represent the storage of data more abstractly as a mapping from attribute identifiers to attribute values.

CONTEXT *META_DATA*
SETS *Attr_id* ; *Attr_value*
END

Pages Context.

The pages of CDIS are device-independent representations of what can be displayed on a screen. Each page is associated with a page number, and each page consists of its contents.

CONTEXT *PAGE_CONTEXT*
SETS *Page_number* ; *Page_contents*
END

Displays Context.

At this abstract level, we model the physical devices with which the users interact with the system. However, we only need to acknowledge that each position is uniquely identified (by its *EDD_id*), each user position has a type, and each user position has a physical display. Some user positions are 'editors' which have the capability of manipulating data and pages.

CONTEXT *DISPLAY_CONTEXT*
SETS *EDD_id* ; *EDD_type* ; *EDD_display*
CONSTANTS *EDDs* , *EDIT* , *EDITORS*
PROPERTIES
$EDIT \in EDD\_type \wedge$
$EDDs \in EDD\_id \rightarrow EDD\_type \wedge$
$EDITORS \subseteq EDD\_id \wedge$
$EDITORS = EDDs^{-1}[\{EDIT\}]$
END

Merge Context.

By merging the previous three contexts (via a **SEES** clause), we can declare a function that can determine the actual display, given the appropriate information. In declaring this function, we use an unfamiliar syntax. In [EB06], we have proposed the introduction of a record-like structure to Event-B. This proposal does not require any changes to the semantics of Event-B, but it gives us a succinct way to define structured data. The declaration of *Disp_interface* in the **SETS** clause of the following context is an example of our proposed syntax

CONTEXT *MERGE_CONTEXT*
SEES *META_DATA* , *DISPLAY_CONTEXT* , *PAGE_CONTEXT*
SETS *Disp_interface* :: *data* : *Attr_id* $\rightarrow$ *Attr_value*,
contents : *Page_contents*
CONSTANTS *disp_values*
PROPERTIES $disp\_values \in Disp\_interface \rightarrow EDD\_display$

The type *Disp_interface* is a record comprising two fields *data* (of type $Attr\_id \rightarrow Attr\_value$) and *contents* (of type *Page_contents*). This record type defines the interface to the function *disp_values*. The intention is that, given a database of values and the device-independent representation of a display, *disp_values* calculates what is actually displayed (i.e. it returns a value of type *EDD_display*). The benefit of using a record type is that it can be refined by adding extra fields (see [EB06] for more details). This is necessary because the actual display is dependent on parameters that are introduced during the refinement stages. The extension of record types through refinement allows us to modify the interface accordingly (an example of this is given in Section 2.4.3).

As in the original CDIS specification, the fact that we represent *disp_values* so abstractly does not undermine the value of the specification. The dynamic part of the specification (shown below) focuses on updating attributes and pages, and defines the pages selected at user positions.

The Abstract Model: A Generic Display

The variable *database* represents the stored data, and *page_selections* records the page number currently selected at a user position. Note that this is a partial function which means that user positions are not obliged to display a page. The variable *pages* is a partial function mapping page numbers and page contents. The variable *private_pages* holds the page contents of a page prior to release. This is intended to model an editor's ability to construct new pages before they are made public. Finally, *trq* models the 'timed release queue' that enables a new version of a

page to be stored until a given time is reached, whereupon it is made public.

MACHINE *ABS_DISPLAY*
SEES
META_DATA, DISPLAY_CONTEXT, PAGE_CONTEXT, MERGE_CONTEXT
VARIABLES *database , pages , page_selections , private_pages , trq*
DEFINITIONS
inv $\widehat{=}$
$database : Attr\_id \rightarrow Attr\_value \wedge$
$pages : Page\_number \nrightarrow Page\_contents \wedge$
$page\_selections : EDD\_id \nrightarrow Page\_number \wedge$
$private\_pages : Page\_number \nrightarrow Page\_contents \wedge$
$trq : Page\_number \nrightarrow Page\_contents \wedge$
$\mathsf{ran}(page\_selections) \subseteq \mathsf{dom}(pages)$
INVARIANT *inv*
INITIALISATION *database , pages , page_selections , private_pages , trq* : (*inv*)

Note that, in addition to type information, the invariant insists that pages can be selected only if they have contents. We keep the model simple by initialising the system to be any state in which the invariant holds.

Almost all of the operations given below correspond to operations defined in the original VDM specification. One exception is the **VIEW_PAGE** operation that uses the *disp_values* function to output an actual display. This is a departure from the original VDM specification but, since outputs must be preserved during refinement, it forces us to ensure that the appearance of actual displays is preserved.

UPDATE_DATABASE models the automatic update of data via the stream of data coming from the airports, and **SET_DATA_VALUE** models the manual update of values (by editors). **DISPLAY_PAGE** enables any user to select a new page to be displayed, and **DISMISS_PAGE** removes a page selection. **RELEASE_PAGE** makes a private page public, and **DELETE_PAGE** enables an editor to delete the contents of a page. In addition to the manual release of pages (via **RELEASE_PAGE**), pages can be released automatically at specific times. **RELEASE_PAGES_FROM_TRQ** models the timed release of pages. However, at this stage no notion of time exists in the specification. Therefore, this operation selects an arbitrary subset of the pages from *trq* to be released. This is refined when we introduce a notion of time (as shown in Section 2.4.3). The operations use common B operators such as function overriding $\lhd$, domain subtraction $\lhd$, and range subtraction $\rhd$.

UPDATE_DATABASE (*ups*) $\widehat{=}$
  PRE
    $ups \in Attr\_id \nrightarrow Attr\_value$
  THEN
    $database := database \lhd\!\!+ ups$
  END ;

SET_DATA_VALUE (*ei* , *ai* , *av*) $\widehat{=}$
  PRE
    $ei \in EDD\_id \wedge$
    $ai \in Attr\_id \wedge av \in Attr\_value$
  THEN
    WHEN $ei \in EDITORS$ **THEN**
      $database (ai) := av$
    END
  END ;

DISPLAY_PAGE (*ei* , *no*) $\widehat{=}$
  PRE
    $ei \in EDD\_id \wedge no \in Page\_number$
  THEN
    WHEN $no \in \mathsf{dom} (pages)$ **THEN**
      $page\_selections (ei) := no$
    END
  END ;

DISMISS_PAGE (*ei*) $\widehat{=}$
  PRE $ei \in EDD\_id$ **THEN**
    WHEN
      $ei \in \mathsf{dom} (page\_selections)$
    THEN
      $page\_selections :=$
        $\{ ei \} \lhd page\_selections$
    END
  END ;

$ed \longleftarrow$ **VIEW_PAGE** (*ei*) $\widehat{=}$
  PRE $ei \in EDD\_id$ **THEN**
    ANY *di* **WHERE**
      $ei \in \mathsf{dom} (page\_selections) \wedge$
      $di \in Disp\_interface \wedge$
      $data (di) = database \wedge$
      $contents (di) =$
        $pages (page\_selections (ei))$
    THEN
      $ed := disp\_values (di)$
    END
  END

RELEASE_PAGE (*no*) $\widehat{=}$
  PRE $no \in Page\_number$ **THEN**
    WHEN
      $no \in \mathsf{dom} (private\_pages)$
    THEN
      $pages (no) :=$
        $private\_pages (no) \parallel$
      $private\_pages :=$
        $\{ no \} \lhd private\_pages$
    END
  END ;

RELEASE_PAGES_FROM_TRQ $\widehat{=}$
  ANY *SS* **WHERE**
    $SS \in$
      $Page\_number \nrightarrow Page\_contents \wedge$
    $SS \subseteq trq$
  THEN
    $pages := pages \lhd\!\!+ SS \parallel$
    $trq := trq - SS$
  END ;

DELETE_PAGE (*ei* , *no*) $\widehat{=}$
  PRE
    $ei \in EDD\_id \wedge$
    $no \in Page\_number$
  THEN
    WHEN $ei \in EDITORS$ **THEN**
      $pages := \{ no \} \lhd pages \parallel$
      $private\_pages :=$
        $\{ no \} \lhd private\_pages \parallel$
      $trq := \{ no \} \lhd trq \parallel$
      $page\_selections :=$
        $page\_selections \rhd \{ no \}$
    END
  END ;

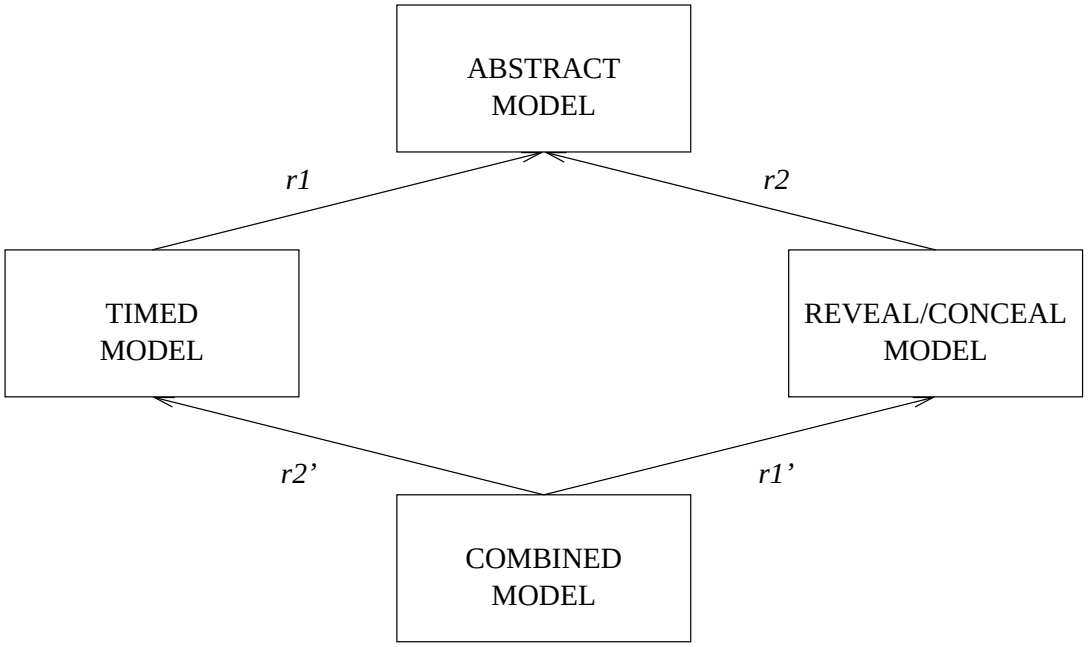


Figure 2.3: Non-linear refinement

2.4.3 Refinement

The abstract specification described in the previous section omitted many of the features that characterise CDIS. However, this made it possible to give a broad overview of the system, including its state variables and operations, within a few pages. Now we use this specification as a basis for refinement in which the omitted details are introduced.

One thing that becomes apparent when using this approach is the lack of order which dictates the sequence of refinements. In the case of CDIS, the way that information recorded in the database is displayed depends on several (unrelated) things. For example, values are coloured according to their age. By adding a notion of time, via refinement, it is possible to model such behaviour. In addition, the model is refined by adding a reveal/conceal feature that enables the air traffic controllers to display or remove 'page overlays' from a screen. The order in which these features are introduced to the specification is arbitrary since one feature does not depend on the other. Enforcing a linear refinement policy in which one feature had to be introduced after the other would be inconvenient because it would be easier to understand a refinement that adds a reveal/conceal feature to the abstract model (see Section 2.4.3) rather than add it to the timed model shown in Section 2.4.3. Similarly the timed model of Section 2.4.3 is easier to appreciate without the reveal/conceal feature of Section 2.4.3. It would be desirable, therefore, to refine the abstract model in two separate ways and then combine the results. This is depicted in Figure 2.3. Note that the refinements introducing time ($r1$ and $r1'$) are not identical, but are closely related (similarly for $r2$ and $r2'$).

Adding Time

In terms of the CDIS subset, there are two main reasons for adding time: each piece of airport data has an age which affects how it is displayed, and the version of each page that is displayed is also time-dependent. In this refinement we shall once again use our proposed syntax for record types [EB06].

Time Context.

We begin by introducing a new context to the development. The set *Date_time* represents all of the different points in time. We also include a total ordering relation (*leq*) between these points.

CONTEXT *TIME*
SETS *Date_time*
CONSTANTS *leq*
PROPERTIES
$leq \in Date\_time \leftrightarrow Date\_time \wedge$
$\forall (a).(a : Date\_time \Rightarrow (a, a) : leq) \wedge$
$\forall (a, b).(a : Date\_time \wedge b : Date\_time \Rightarrow$
$((a, b) : leq \wedge (b, a) : leq \Rightarrow a = b) \wedge$
$((a, b) : leq \vee (b, a) : leq)) \wedge$
$\forall (a, b, c).(a : Date\_time \wedge b : Date\_time \wedge c : Date\_time \Rightarrow$
$((a, b) : leq \wedge (b, c) : leq \Rightarrow (a, c) : leq))$
END

Meta Data Context.

In order to record the age of a piece of data as well as its value, we refine the *META_DATA* context by defining a record type *Attrs* with two fields *value* and *last_update*.

CONTEXT *META_DATA1*
SEES *META_DATA* , *TIME*
SETS *Attrs* :: *value* : *Attr_value*,
last_update : *Date_time*
END

Note that the range of *value* is of our original value type *Attr_value*. The gluing invariant of the refined model will ensure that the values of the entries in the refined database will match the corresponding entries in the original database. (This technique of 'wrapping' an abstract type in a refined type occurs frequently in our approach.) The field *last_update* (of type *Date_time*) records the time at which the value of the attribute was last updated.

Pages Context

We proceed by refining the pages context in a similar manner. We declare a record type *Page* with two fields: *page_contents* holds the structure of a page, and *creation_date* holds the time at which a page was created. Note that this has nothing to do with the time at which the page is released. In order to model the timed release queue faithfully, we must associate a release date with every page on the queue. By using our proposed syntax for record refinement [EB06], this is achieved by defining a subtype of *Page* (called *Rel_page*) whose elements have an additional field called *release_date*.

CONTEXT *PAGE_CONTEXT1*
SEES *TIME* , *PAGE_CONTEXT*
SETS
Page :: *page_contents* : *Page_contents*,
creation_date : *Date_time* ;
Rel_page SUBTYPES *Page* WITH *release_date* : *Date_time*
END

Only pages of type *Rel_page* occur on the timed release queue. We shall see how the refinement of the operation **RELEASE_PAGES_FROM_TRQ** uses this additional information.

Merge Context.

Now that we have introduced a notion of time, the display function *disp_values* can be augmented so that the ages of the data in the database is taken into account when they are displayed. We change the interface of the function by adding a new field to *Disp_interface* called *time*. The operator 'EXTEND' is similar to the 'SUBTYPES' operator, but it adds fields to *all* elements of the record type.

CONTEXT *MERGE_CONTEXT1*
SEES
META_DATA , *DISPLAY_CONTEXT* , *PAGE_CONTEXT* ,
TIME , *META_DATA1*, *PAGE_CONTEXT1* , *MERGE_CONTEXT*
SETS EXTEND *Disp_interface* WITH *time* : *Date_time*
END

Whenever the function *disp_values* is called, the current time can be passed as a parameter so that the ages of the relevant data can be determined. In CDIS, the colour of a value when displayed indicates its age (although this detail is not included at this level of abstraction).

The Refined Model: A Timed Display.

The state variables and the operations of *ABS_DISPLAY* are refined to incorporate the timed context. Four of the variables in the refinement replace those of the abstract model. The invariant gives the relationship between these concrete variables and their abstract counterparts. For example, the abstract variable *database* is refined by *timed_database*, and they are related because the attribute values held in *database* can be retrieved from the *value* fields in *timed_database*.

REFINEMENT *ABS_DISPLAY1*
REFINES
ABS_DISPLAY
SEES
META_DATA, *DISPLAY_CONTEXT*, *PAGE_CONTEXT*, *MERGE_CONTEXT*,
TIME , *META_DATA1* , *PAGE_CONTEXT1* , *MERGE_CONTEXT1*
VARIABLES
timed_database ,
page_selections ,
timed_pages ,
private_timed_pages ,

dated_trq ,
time_now
DEFINITIONS
inv1 $\widehat{=}$
$timed\_database \in Attr\_id \rightarrow Attrs \wedge$
$timed\_pages \in Page\_number \nrightarrow Page \wedge$
$private\_timed\_pages \in Page\_number \nrightarrow Page \wedge$
$dated\_trq \in Page\_number \nrightarrow Rel\_Page \wedge$
$time\_now \in Date\_time \wedge$
$database = (timed\_database ; value) \wedge$
$\mathsf{ran}\ (page\_selections) \subseteq \mathsf{dom}\ (timed\_pages) \wedge$
$pages = (timed\_pages ; page\_contents) \wedge$
$private\_pages = (private\_timed\_pages ; page\_contents) \wedge$
$trq = (dated\_trq ; page\_contents) \wedge$
$\forall n . (n \in \mathsf{dom}\ (timed\_pages) \Rightarrow$
$(creation\_date (timed\_pages (n)), time\_now) \in leq) \wedge$
$\forall n . (n \in \mathsf{dom}\ (private\_timed\_pages) \Rightarrow$
$(creation\_date (private\_timed\_pages (n)), time\_now) \in leq) \wedge$
$\forall n . (n \in \mathsf{dom}\ (dated\_trq) \Rightarrow$
$(creation\_date (dated\_trq (n)), time\_now) \in leq)$
INVARIANT *inv1*

Some of the operations affected by the refinement are shown below.

UPDATE_DATABASE (*ups*) $\widehat{=}$
PRE $ups \in Attr\_id \nrightarrow Attr\_value$ **THEN**
ANY *ff* **WHERE**
$ff \in Attr\_id \nrightarrow Attrs \wedge$
$\mathsf{dom}\ (ff) = \mathsf{dom}\ (ups) \wedge$
$(ff ; value) = ups \wedge$
$(ff ; last\_update) = \mathsf{dom}\ (ff) \times \{ time\_now \}$
THEN
$timed\_database := timed\_database \leftarrow ff$
END
END

The parameter to the **UPDATE_DATABASE** operation maintains its type, but the **ANY** clause is used to construct a new mapping from *Attr_id* to *Attrs* all of whose *last_update* components are assigned to the current time (to reflect the time of the update). This mapping is used to overwrite the appropriate entities in the timed database. An interesting refinement occurs in the operation **RELEASE_PAGES_FROM_TRQ**. Rather than selecting an arbitrary subset of *trq* to release, *time_now* is used to select those elements whose release date is earlier than the current time. The released pages (held in *timed_pages*) are updated accordingly.

RELEASE_PAGES_FROM_TRQ $\widehat{=}$
LET *SS* **BE** $SS =$
$dated\_trq \rhd \{ rp \mid rp \in Rel\_Page \wedge (release\_date (rp) , time\_now) \in leq \}$
IN
$timed\_pages := timed\_pages \leftarrow SS \parallel$
$dated\_trq := dated\_trq - SS$
END

Next, we introduce a new operation, called **CLOCK** that increases the current time by some unspecified amount. This operation models the passing of time.

CLOCK $\widehat{=}$
ANY *time_next* **WHERE**
time_next $\in$ *Date_time* $\wedge$
(*time_now* , *time_next*) $\in$ *leq* $\wedge$
time_next $\neq$ *time_now*
THEN
time_now := *time_next*
END

A Different Refinement: Page Overlays

Now we consider an alternative refinement to the abstract model in which the reveal/conceal feature is added rather than time. First we augment the relevant context **PAGE_CONTEXT**

MACHINE *PAGE_CONTEXT2*
SEES
PAGE_CONTEXT
SETS
Graphic_background ;
Overlay_Page_contents SUBTYPES *Page_contents* WITH *overlay* : *Graphic_background*
END

This definition adds the field *Overlay_Page_contents* to a subset of all page contents (i.e. those pages that possess an overlay). Now we can augment the model by introducing a new variable *concealed_displays* to record which displays are concealing their overlays.

REFINEMENT *ABS_DISPLAY2*
REFINES
ABS_DISPLAY
SEES
META_DATA , *DISPLAY_CONTEXT* , *PAGE_CONTEXT* ,
PAGE_CONTEXT1 , *MERGE_CONTEXT*
VARIABLES
database ,
pages ,
page_selections ,
private_pages ,
trq ,
concealed_displays
DEFINITIONS
inv2 $\widehat{=}$
concealed_displays $\subseteq$ *EDD_id* $\wedge$
concealed_displays $\subseteq$ dom (*page_selections*)
INVARIANT
inv2

In a similar manner to the timed refinement, the operations of the abstract model are refined to incorporate reveal/conceal behaviour. In this case we introduce operations to model the users' ability to toggle between revealing and concealing overlays on their displays. It is not

important to understand the details of the operation **RELEASE_PAGES_FROM_TRQ**, but it is important to note the structure of the operation because it will influence the structure of the corresponding operation in the combined model of Section 2.4.3.

RELEASE_PAGES_FROM_TRQ $\widehat{=}$
ANY *SS* **WHERE**
$SS \in Page\_number \nrightarrow Page\_contents \wedge$
$SS \subseteq trq$
THEN
LET *p_nums* **BE**
$p\_nums = \{ no \mid no \in \mathsf{dom}(SS) \wedge no \in \mathsf{dom}(pages) \wedge$
$pages(no) \in Overlay\_Page\_contents \wedge$
$SS(no) \notin Overlay\_Page\_contents \}$
IN
LET *edd_ids* **BE**
$edd\_ids = \mathsf{dom}(page\_selections \rhd p\_nums)$
IN
$pages := pages \lhd\!\!+ SS \parallel$
$trq := trq - SS \parallel$
$concealed\_displays := concealed\_displays - edd\_ids$
END
END
END ;

TOGGLE_REVEAL_CONCEAL (*ei*) $\widehat{=}$
PRE $ei \in EDD\_id$ **THEN**
SELECT $ei \in concealed\_displays$ **THEN**
$concealed\_displays := concealed\_displays - \{ ei \}$
END
END ;

TOGGLE_REVEAL_CONCEAL2 (*ei*) $\widehat{=}$
PRE $ei \in EDD\_id$ **THEN**
SELECT
$ei \in \mathsf{dom}(page\_selections) \wedge$
$pages(page\_selections(ei)) \in Overlay\_Page\_contents \wedge$
$ei \notin concealed\_displays$
THEN
$concealed\_displays := concealed\_displays \cup \{ ei \}$
END
END

Putting the Refinements Together

Now that we have two distinct refinements of the same abstract model, the next step is to combine them into a single model that refines both of them (as in the COMBINED MODEL depicted in Figure 2.3). We do this in a systematic way to promote the idea that the combination of non-linear refinements could be done automatically.

The combined context is simply the amalgamation of the sets, constants and properties of the contexts *PAGE_CONTEXT1* and *PAGE_CONTEXT2*. This can be achieved via the **SEES** clause, but for completeness we show this union as a third context *COMBINED_PAGE_CONTEXT*.

CONTEXT *COMBINED_PAGE_CONTEXT*
SEES *TIME* , *PAGE_CONTEXT*
SETS
Page :: *page_contents* : *Page_contents*,
creation_date : *Date_time* ;
Rel_page SUBTYPES *Page* WITH *release_date* : *Date_time* ;
Graphic_background ;
Overlay_Page_contents SUBTYPES *Page_contents* WITH *overlay* : *Graphic_background*
END

The model is constructed by amalgamating the variables of both refined models and taking the conjunction of their invariants. Note that the consistency of the individual refinements is preserved by the combination because the variables of the separate refinements are distinct. Although proof obligations are still generated by the tool, the proofs follow those of the individual refinements. Future work aims to show that such combinations can be done without the need to generate further proof obligations. In other words, consistency is preserved by this procedure.

REFINEMENT *COMBINED_ABS_DISPLAY*
REFINES
ABS_DISPLAY
SEES
META_DATA, *DISPLAY_CONTEXT*, *PAGE_CONTEXT*, *MERGE_CONTEXT*,
TIME , *META_DATA1* , *PAGE_CONTEXT1* , *MERGE_CONTEXT1*
VARIABLES
timed_database ,
page_selections ,
timed_pages ,
private_timed_pages ,
dated_trq ,
time_now ,
concealed_displays
INVARIANT
inv1 $\wedge$ *inv2*

Combining the operations is a bit more tricky, but typically this consists of taking the conjunction of (i.e. strengthening) the guards, and composing the bodies of the operations via the parallel operator $\|$.

RELEASE_PAGES_FROM_TRQ $\widehat{=}$
LET *SS* **BE** *SS* =
dated_trq $\rhd$ { *rp* | *rp* $\in$ *Rel_Page* $\wedge$ (*release_date* (*rp*) , *time_now*) $\in$ *leq* }
IN
LET *p_nums* **BE**
p_nums = { *no* | *no* $\in$ dom (*SS*) $\wedge$ *no* $\in$ dom (*timed_pages*) $\wedge$
timed_pages (*no*) $\in$ *Overlay_Page_contents* $\wedge$
SS (*no*) $\notin$ *Overlay_Page_contents* }
IN
LET *edd_ids* **BE**
edd_ids = dom (*page_selections* $\rhd$ *p_nums*)
IN
timed_pages := *timed_pages* $\lhd\!\!+$ *SS* ||
dated_trq := *dated_trq* $-$ *SS* ||
concealed_displays := *concealed_displays* $-$ *edd_ids*
END
END
END

It can be seen that this operation combines the features of the corresponding operations in the individual refinements.

2.5 Case Study 5: Ambient Campus

Mobile agent systems are increasingly attracting attention of software engineers. However, issues related to fault tolerance and exception handling in such systems have not yet received the level of attention they deserve. In particular, formal support for validating the correctness and robustness of fault tolerance properties is still under-developed. Within the Ambient Campus case study, we developed an initial approach to dealing with such issues in the context of a concrete system for dealing with mobility of agents (CAMA), and a concrete technique for verifying their properties (partial order model checking). An overall goal of this strand of our work is a formal model for the specification, analysis and model checking of CAMA designs. To achieve it, we use process algebras and high-level Petri nets.

In concrete terms, our approach reported in [IKKR05] is first to give a formal semantics (including a compositional translation) of a suitably expressive subset of CAMA in terms of an appropriate process algebra and its associated operational semantics. The reason why we chose a process algebra semantics is twofold: (i) process algebras, due to their compositional and textual nature, are a formalism which is very close to the actual notations and languages used in real implementations; and (ii) there exists a significant body of research on the analysis and verification of process algebras. In our particular case, there are two process algebras which are directly relevant to CAMA, viz. KLAIM [DNLM05, DNFP98] and π-calculus [Par01], and our intention is to use the former as a starting point for the development of the formal semantics.

The process algebra semantics of CAMA can then be used as a starting point for developing efficient model checking techniques aimed at verifying the behavioural correctness of CAMA designs. In our approach, we are specifically interested in model checking techniques which alleviate the state space explosion problem, and for this reason we adopted a partial order model checking based on Petri net unfoldings [Kho03]. To be able to use it, we will take advantage of

a semantics preserving translation from the process terms used in the modelling of CAMA to a suitable class of high-level Petri nets based on [DKK06b, DKK06a].

CAMA is a middleware supporting rapid development of mobile agent software. It offers a programmer a number of high-level operations and a set of abstractions which help to develop multi-agent applications in a disciplined and structured way. CAMA is an extensible system. Its inter-agent communication is based on the LINDA paradigm which provides a set of language-independent coordination primitives that can be used for coordination of several independent pieces of software. They allow processes to put *tuples* (vectors of values) in a shared tuple space, remove them, and test for their presence. Input operations use special tuples called *templates*, where some fields are replaced with wildcards that can match any value.

The approach taken in the Ambient Campus case study is based on the asymmetric model of agent systems within the *location-based* paradigm. The main part of communication and control is implemented by a dedicated service, called *location*. The approach supports large-scale mobile agent networks in a predictable and reliable manner. Moreover, location-based architecture eliminates the need for employing complex distributed algorithms, such as voting or agreement.

Chapter 3

Discussion of issues

3.1 Towards an Algebra of Abstractions for Communicating Processes

It is often desirable to describe the interface of an implementation system at a different (usually more detailed) level of abstraction to the interface of the relevant specification. This calls for a relation aimed at formalising the notion that a process is an acceptable implementation of another target process in the event that they possess different interfaces. Let us consider a *specification* network, P_{net}, composed of n communicating processes $P_1, \ldots, P_n$, and a corresponding *implementation* network, Q_{net}, also composed of n processes, $Q_1, \ldots, Q_n$. Intuitively, P_i is intended to be Q_i's specification; we shall also refer to P_i as a *target* or *base* system, and to Q_i as a *source* or *implementation* system. Note that although P_i's and Q_i's interfaces need not coincide, we do assume that the interface of Q_{net} at the boundary with the external environment is the same as that of P_{net} (see Figure 3.1), and that we are not interested in the details of inter-process communication within the networks P_{net} and Q_{net}. In the CSP notation, this means that the specification network P_{net} is of the form $(P_1 \| P_2 \| \cdots \| P_n) \setminus A$, where A is the set of actions used for the internal communication by the P_i's, while the implementation network is of the form $Q_{net} = (Q_1 \| Q_2 \| \cdots \| Q_n) \setminus B$, where B is the set of actions used for the internal communication by the Q_i's.

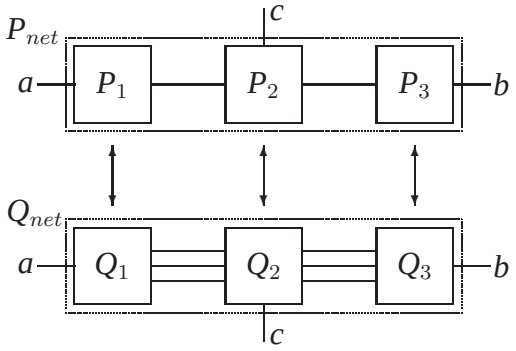


Figure 3.1: Network connectivity where pairs of corresponding processes, Q_i and P_i, may have different observable actions.

In the usual treatment of process algebras, such as [Mil89, Ros98], the notion that Q_{net}

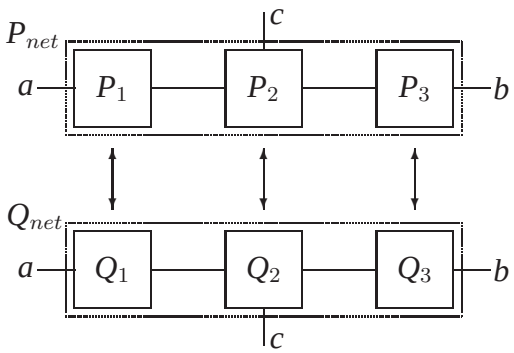


Figure 3.2: Network connectivity where each Q_i has the same observable actions as P_i.

implements P_{net} is based on the idea that Q_{net} is more deterministic than (or equivalent to) P_{net} in terms of the chosen semantics. In practice, to formally verify that such a property holds, one can proceed in either of the following two ways.

- The first approach is to compare P_{net} and Q_{net} according to a chosen notion of refinement or equivalence. This is a straightforward approach, but one which potentially suffers from a severe state space explosion, since the compared processes are obtained by combining n components.

- The second, usually much better, approach attempts to (i) show an appropriate refinement or equivalence holds for each pair of processes P_i and Q_i, and (ii) resort to general theorems to infer the desired relation between the complete networks, P_{net} and Q_{net}.

Within the standard approaches, such as [Mil89, Ros98], the latter *compositional* way of proving correctness of the implementation network is handled *only* in the case when for each i, P_i and Q_i have the same actions in their interfaces. Hence the networks in Figure 3.1 cannot really be treated by this technique, as the constituent processes should be like in Figure 3.2. Yet in deriving an implementation from a specification we will often wish to implement abstract, high-level *interface* actions at a lower level of detail and in a more concrete manner. For example, the link available to connect base components P_i, P_j may be unreliable, and so may need to be implemented, in the sources Q_i, Q_j, by a pair of channels, one for data and one for acknowledgements. Or an intended implementation $\overline{Q_i}$ of P_i may be liable to fail itself, so that the final implementation Q_i is built by assembling redundant replicas of $\overline{Q_i}$, and thus has each channel of P_i replicated [Lam78] (such a scenario was one of the original motivations behind the work [KMP97] of which this paper is a continuation). Or it may simply be the case that a high-level action of P_i is rendered in a more concrete, and hence more implementable, form. As a result, the interface of an implementation process may exhibit a lower (and so different) level of abstraction than a specification process.

In the process algebraic context, dealing with *interface difference* necessitates the development of what Rensink and Gorrieri [RG01] have termed a *vertical implementation* relation. This should adequately capture the nature of the relationship between a specification and an implementation whose interfaces differ; and should collapse into the standard, *horizontal* one whenever the two interfaces happen to coincide. In works [KMP97, BKPPK02, BKP04], we independently identified this 'collapsing' requirement as *accessibility* or *realisability*, effectively pioneering it within the CSP process model [Ros98].

Technically, in our preceding works [BKPPK02, BKP04], processes are formalised using the CSP language, with its standard failures-divergences semantics [Ros98]. The implementation relation is formulated in terms of failures and divergences of the implementation and target processes. Interface difference is modelled by endowing the implementation relation with parameters called extraction patterns. These are intended to interpret implementation behaviour as target behaviour (translating, in particular, traces), and suitably constrain the former in connection to acceptable refusals.

We developed the theory of [BKPPK02, BKP04] under that crucial restriction of the *one-to-one* communication paradigm. That is, we assumed no communication action is shared by more than two processes within a distributed network of processes (as is the case in Figure 3.1). This, in particular, excluded systems with broadcast, as well as disallowed, say, a potentially complex group protocol at the implementation level to be abstracted into a single action at the specification level. Another limitation of [BKPPK02, BKP04] was that it only established the implementation relation to distribute over network composition (i.e. compositionality), not other useful operations employed to combine and construct processes, notably choice.

In the current work, reported in [KPPK06], no restrictions are now placed on the class of specification processes allowed, other than a quite natural divergence-freedom requirement. Realisability holds in its purest form, in the sense that implementation in the absence of interface difference collapses into standard CSP refinement. Moreover, we can now deal with process networks where the one-to-one communication constraint need not be adhered to; this allows group communication to be modelled. We also generalise compositionality results, through a treatment intended to show that the implementation relation distributes over the main CSP operators, beginning from internal choice. Such results are apt to prove beneficial in the compositional verification of systems. It is worth noting, in particular, that more general scenarios can now be handled than process network topologies as in Figure 3.1, where implementation and base processes are paired in a unique way. For example, in a situation as in Figure 3.3 we can apply new results to prove that Q_{net} implements P_{net} in a stepwise fashion, by showing that: (i) Q_1 implements P_1, and (ii) the parallel composition of Q_2 and Q_3, with their mutual interaction hidden, implements P.

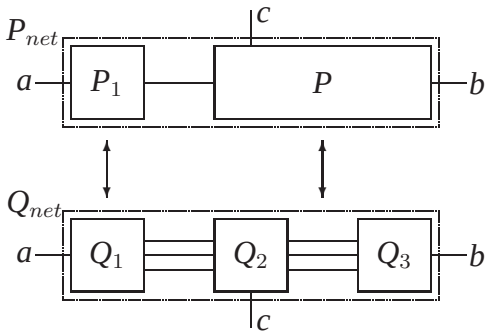


Figure 3.3: Networks with different topology.

3.2 Rigorous Development of Fault-Tolerant Agent Systems

Mobile agent systems are complex distributed systems that are dynamically composed of independent agents. Usually agents are designed by different developers to perform individual

computational tasks. The agent technology naturally solves the problem of partitioning complex software into smaller parts that are easier to analyse, design and maintain. However, to ensure interoperability of agents, the individual developments should adhere to a certain "standard", which would guarantee compatibility of constructed agents yet avoid over-constraining the development process. Middleware supporting agent execution is also distributed. In this work we show how to formally develop a distributed middleware in such a way that its different parts can be taken apart and implemented independently [ILRT06].

The initial results of this work were applied in the development of the CAMA (Context-Aware Mobile Agents) middleware system [AIR06, Ili06]. The core part of the middleware – the scoping mechanism [IR05] – was formally designed and then implemented according to the resultant specification [ILRT05]. For other parts, such as disconnection toleration and agent recovery results of the formal development helped to re-engineer the middleware implementation.

We start from an abstract specification of the overall agent system, i.e., abstractly model agents together with the location supporting inter-agent communication. In a number of correctness preserving steps we incorporate various system properties, including fault tolerance, into the specification. Finally, we arrive at the specification of entire middleware, which can be decomposed into parts to be implemented by the location and by each individual agent.

In the independent development of individual agents the programmers merely need to augment this abstract part with an implementation of the desired agent functionality. Such an approach allows us to ensure inter-operability of individually developed agents and the correctness of the overall system. Moreover, since the proposed patterns contain abstract specifications of the means for detecting agent failures, such as disconnections and crashes, and the corresponding error recovery procedures, we can guarantee fault tolerance of an agent system developed according to the proposed approach.

One of the major challenges in designing agent systems lies in ensuring interoperability of agents. This problem can only be properly addressed if we define the essential properties of the overall agent system, derive the properties to be satisfied by the location and each agent, and ensure that they are preserved in the agent and location development. This goal can be achieved by adopting the system approach to developing agent systems, i.e., modelling the entire set of agents together with the location that provides the infrastructure for agent communication.

3.2.1 Fault-tolerance

One of the essential requirements of multi-agent systems is the ability to operate in a volatile, error prone environment. Hence we aim at developing fault tolerant agent systems, i.e., systems which can withstand various kinds of faults. The most typical class of faults in our case is a temporal loss of connection. It might cause errors or delays in communication between cooperating agents. In our first refinement step we introduce an abstract representation of this type of fault.

In most cases an agent loses connection only for a short period of time. After connection is restored, the agent is willing to continue its activities virtually uninterrupted. Therefore, after detecting connection loss, the location should not immediately disengage the disconnected agent but rather set a deadline before which the agent should reconnect. If the disconnected agent restores its connection before the deadline then it can continue its normal activity. However, if the agent fails to do it, the location should disengage the agent.

This behaviour can be formally modelled by the timeout mechanism. Upon detecting disconnection the location activates a timer. If the agent reconnects before the timeout then the timer is stopped. Otherwise, the location forcefully disengages the disconnected agent.

During the formal development of the system we use a combination of the superposition refinement and atomicity refinement. With atomicity refinement a single event is refined into a set of events. A simple abstract event can be represented as of alternative events and this allows use to explicitly introduce normal behaviour and recovery actions.

The net direction of refinement is a providing a finer recovery actions for agent failures. Initially, any agent failure is treated as an unrecoverable error. Upon detecting an error, the failed agent is removed from the scope and disengaged from the location. In our next refinement step we distinguish between recoverable and unrecoverable errors. Namely, upon detecting an error the agent at first tries to recover from it (probably involving some other agents into the error recovery). If the error recovery eventually succeeds then the normal operational state of the agent is restored. Otherwise, the error is treated as unrecoverable.

While specifying error recovery procedures, it is crucial to ensure that the error recovery terminates, i.e., does not continue forever. To ensure this, we introduce the variable which limits the amount of error recovery attempts for each agent. Each attempt of error recovery decrements this value by one. When for some agent the recovery limit becomes zero then agent error recovery terminates and the error is treated as unrecoverable.

3.2.2 Interoperability

At the initial stages of the system development we mainly focused on modelling interactions of agents with the location. We proceed by introducing an abstract representation of the scopes as an essential mechanism which governs agent interactions while they are involved in cooperative activities.

The scoping mechanism has a deep impact on modelling error recovery in agent systems. For instance, if a scope owner irrecoverably fails, then, to recover the system from this error, the location should close the affected scope and force all agents to leave.

Each scope provides the isolated coordination space for compatible agents to communicate. Compatibility of agents is defined by their roles – abstract descriptions of agent functionality. To ensure compatibility of agents in a scope, each scope supports a certain predefined set of roles. When an agent joins a scope, it chooses one of the supported roles. We assume that an agent can join a scope only in one role and this role remains the same while the agent is in the scope. However, an agent might leave a scope and join it in another role later.

The creator of the scope defines the minimal and maximal numbers of agents that are allowed to play each supported role. This is dictated by the logical conditions on the scope functionality. For instance, if the scope is created for purchasing a certain item on an electronic auction then there are must be only one seller and at least one buyer for a scope to function properly.

However, agent systems are asynchronous systems. Therefore, at the time of scope creation it cannot be guaranteed that agents will take all the required roles in the right proportions at once and the scope will instantly become functional. Since agents join and leave the scope arbitrarily, the scope can be in various states at different instances of time: *pending*, when the number of agents is still insufficient for normal functioning of the scope; *expanding*, when the

scope is functional but new agents can still join it; *closed*, when the maximal allowed number of agents per each role is reached.

3.2.3 Conclusion

In our development we adapted the system approach, i.e., captured the behaviour of agents together with their communication environment. While carrying out the development of the system by refinement, we modelled the essential properties of agent systems and incorporated fault tolerance mechanisms into the system specification. We demonstrated how to define the mechanisms for tolerating agent disconnections typical for mobile systems as well as agent crashes.

The proposed approach provides the developers of agent systems with a formal basis for ensuring inter-operability of independently developed agents. Indeed, by decomposing the proposed formal model of the middleware into the parts to be implemented by the agents and by the location and ensuring adherence of their implementations to these specifications, we can ensure agent inter-operability.

3.3 Deriving specifications

The research on "Deriving Specifications" was reported on in §2.2.1 of (D9) the *Preliminary report on methodology*.[1] The "HJJ approach" (after the initial letters of the family names of the three authors) was first set out in [HJJ03]. Not only has this research continued, it is clear that it is attracting significant attention. We will not report the technical details here since adequate material can be cited.

As an update on the state as in D9, we can report

- Ian Hayes (University of Queensland) presented the HJJ method at the REFT workshop associated with FM-06.
- Cliff Jones based his keynote talk at DSVIS-05 on the HJJ method; in particular, he addressed the application of the method to those systems which include human players (see [Jon05d]).
- Joey Coleman presented a joint (with Cliff Jones) paper on the ideas at the REFT workshop associated with FM-06 [CJ05].
- Joey Coleman wrote up the REFT presentation as a (solo) paper [Col06].
- Cliff Jones based his invited talk at the IEEE ICECCS-2005 seminar in Shanghai on the HJJ approach [Jon05c]
- A journal submission (see [JHJ06] for a pre-print) has been made by the original three HJJ authors to *Acta Informatica*.
- Cliff Jones has given several seminars on this topic including ones at an IFIP event (VSTTE) in Zurich and FM-E/FACS.

[1] Cliff Jones also described this work to the first Rodin Review in Brussels on 2005-09-30.

- This last talk led to a "rebuttal" talk by Prof Tom Maibuam (Canada) in which he debated several of the technical decisions in HJJ approach (June 2006, London BCS HQ).

There clearly remains research to be done on HJJ. Of particular relevance to Rodin is the fact that the journal paper [JHJ06] stops short of giving a semantics for ways of combining whole specifications.

3.4 Synthesis of Scenario Based Test Cases from B Models

Software models are usually built to reduce the complexity of the development process and to ensure software quality. A software model is an abstraction in the sense that it captures the most important requirements of the system while omitting unimportant details. A model is usually a specification of the system which is developed from the requirements early in the development cycle [DJK+99]. This paper concerns with formal models only; in particular, we deal with model oriented formal languages like Z [Spi88], VDM [Jon90b] and B [Abr96]. By model oriented we mean, the system behaviour is described using an explicit model of the system state along with operations on the state. We will focus on B models only.

Model based testing is usually based on the notion of a coverage graph obtained from the symbolic execution of the model. A subset of the paths in this graph can be treated as a test suite from the viewpoint of test case generation. Even though model based testing is an incomplete activity, the selected behaviours could be made effective in the sense that they capture the interesting activities of the system and hence the success of their testing would give us confidence about its correctness.

Existing testing tools or techniques [BLLP04, SLB05] dealing with model oriented languages partition the input space of the operations into equivalence classes to create operation instances. Then a Finite State Automaton (FSA) or a coverage graph is constructed in which the initial node corresponds to the initial state of the model, edges correspond to application of operation instances. Usually a coverage graph is constructed up to a predefined depth or size. Some paths of this graph are selected as test cases. When the implementation is subjected to the same sequence of operations as in a test case, we get an image of the original path in the model execution. Now if the properties of the implementation path matches with the properties of the path in the model, we declare that the implementation has passed the test case; otherwise, a failure.

However, in these approaches, there is no guarantee that the user scenarios are tested. A user scenario is like a usecase scenario in UML [OMG05]; in this article, we use scenarios and usecases interchangeably. The paths that we test as test cases may bear no resemblance to the operation sequences in relation to user scenarios. How to know that we are not missing out some scenarios? Of course if all possible operation instances do appear in the coverage graph, and we are able to test all of them, we can say that the user scenarios have been tested in an implicit way. But since we fix a predefined bound on the depth of the coverage graph, some operation instances may lie beyond this bound, and then there is no way to locate them. Furthermore, some valid operation instance may not appear in the graph at all. In this paper, we address these issues. Of course, here we assume that the entire development path from the specification to code is not entirely formal, in which case testing may not be necessary; however, in practice, the entire development process is less often formal.

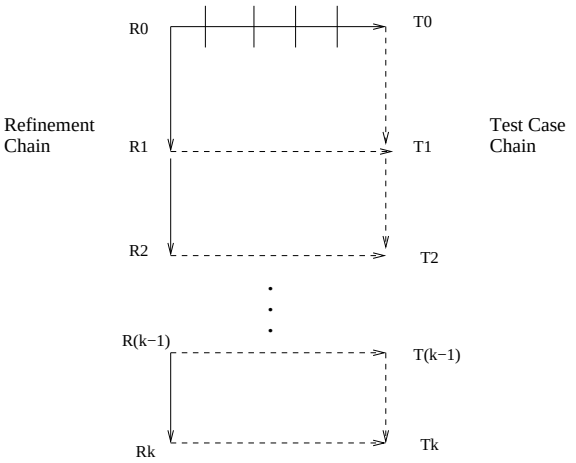


Figure 3.4: The Basic Idea

The basic idea behind our paper can be seen from Figure 3.4. We define an initial usecase-based test case T_0 in terms of a sequence of operations in relation to the initial specification R_0. Thereafter, given any successive refinement pair R_i and R_{i+1}, and T_i as the usecase-based test case for R_i, we derive T_{i+1} such that it is a valid behaviour of R_{i+1}, and in addition, T_i and T_{i+1} are equivalent to each other as far as the original test case is concerned. The main contributions of our paper are:

- We relate our test cases to user scenarios; in the process, we also find out if the refinement has missed out on some scenarios.
- We generate a small number of test cases, and our approach is much more focused. The result is that the time to execute the test cases becomes smaller.

The organization of this section is as follows. Section 3.4.1 discusses the testing terminology we use. Section 3.4.3 describes the problem in a formal manner. In Section 3.4.4, we discuss our approach over a running example. Section 3.4.10 discusses the strengths and the weaknesses of our approach. Section 3.4.11 concludes the material.

3.4.1 Terminology

A *testing criterion* is a set of requirements on test data which reflects a notion of adequacy on the testing of a system [RAO92, ZHM97]. An adequacy criterion serves two purposes: (a) it defines a stopping rule which determines whether sufficient testing has already been done; so, testing can now be stopped, and (b) it provides measurements to obtain the degree of adequacy obtained after testing stopped. For our purpose, the testing criterion would be to test the usecases. However, the usecases are usually generic in nature; so the criterion would be to test some instances of the usecases.

In model based testing, the test cases that are derived from a model always refer to an abstract name space. Since they would be used to test the implementation, it is necessary to define a mapping between the abstract name space of the model and the concrete name space of the implementation. Gannon et al. [GHM97] have termed this as *representation Mapping*. In the context of test oracle generation, there are two types of mappings: control and data [RAO92].

Control mappings are between control points in the implementation and locations in the specification; these are the points where the specification and the implementation states are to be matched. Data mappings are transformations between data structures in the implementation and those in the specification.

3.4.2 Existing Approaches

The work by Dick and Faivre [DF93] is a major contribution to the use of formal methods in software testing. A VDM specification has state variables and an invariant (Inv) to restrict the variables. An operation, say OP, is specified by a pre-condition (OP_{pre}) and a post-condition (OP_{post}). The approach partitions the input space of OP by converting the expression ($OP_{pre} \wedge OP_{post} \wedge$ Inv) into its Disjunctive Normal Form (DNF), and each disjunct, unless a contradiction, represents an input subdomain of OP. Next as many operation instances are created as the number of non-contradictory disjuncts in the DNF. An attempt is then made to create a FSA in which each node represents a possible machine state and an edge represents an application of an operation instance. A set of test cases is then generated by traversing the FSA, each test case being a sequence of operation instances.

BZ-Testing Tool (BZ-TT)[BLLP04] generates functional test cases from B as well as Z specifications. The test case generation proceeds in the following steps.

- Each operation is partitioned into a set of operation instances so that each partition corresponds to exactly one control path within the operation; the conjunction of all predicates along with the postcondition in relation to the control path is called the effect predicate of the operation instance.

- The free state variables in each effect predicate is assigned with their maximum and minimum values to obtain a set of boundary goals. Similarly, boundary input values are obtained by giving maximum and minimum values to the input variables in the effect predicate.

- A preamble is computed by using a Constraint Logic Programming (CLP) Solver which finds a path through symbolic execution from the initial state to a boundary state, a state satisfying a given boundary goal. And then relevant operation instances are applied at the boundary state by giving them boundary inputs.

Satpathy et al. [SLB05] discuss the prototype of a tool called ProTest which performs testing of an implementation in relation to its B model. The tool performs partition analysis using a technique similar to that of Dick and Faivre. A finite coverage graph is created from a symbolic execution of the B model by a model checking tool called ProB [LB05]. Some paths starting from the initial state are taken as test cases. The ProTest tool can run Java programs. So the B model and its implementation are run simultaneously by the tool – the former symbolically and the latter at the concrete level – in relation to a test case and similar model and implementation states are matched to assign a verdict.

All approaches usually partition the input space to create operation instances, but when a specification is further refined, the original partitions may have no meaning in relation to the refinements because the data space might have changed. Derrick and Boiten [DB99] have developed a strategy to transform the operation instances so that they remain meaningful in relation to the appropriate refinement.

3.4.3 The B Method

The B-method, originally developed by J.-R. Abrial [Abr96], is a theory and methodology for formal development of computer systems. B is used to cover the whole range of software development cycle; the specification is used to generate code with a set of refinement steps in between. At each stage, the current refinement needs to be proved consistent with the previous refinement.

The basic unit of specification in the B-method is called a *B machine*. Larger specifications can be obtained by composing B machines in a (tree-like) hierarchical manner. This is a design restriction on B with a view to making the proofs compositional. An individual B machine consists of a set of variables, an invariant to restrict the variables, and a set of operations to modify the state. An operation has a precondition, and an operation invocation is defined only if the precondition holds. The initialization action and an operation body are written as atomic actions coded in a language called the *generalized substitution language* [Abr96]. The language allows specification of deterministic and non-deterministic operations and assignments. An operation invocation transforms a machine state to a new state. The behaviour of a B machine can be described in terms of a sequence of operations, and the first operation of the sequence originates from the initial state of the machine.

3.4.4 The Problem

It should be clear that, in case of the BZ-TT and ProTest approaches, if all operation instances do appear in the coverage graph then, and each such instance is tested, it would imply that all scenarios are covered in an implicit manner. However, usually all such instances do not appear. The reasons are as follows:

- If the model invariant is weak then a valid operation instance may not be reachable. For instance, the constraint $10 \leq X \leq 20$ could be present in a model as $0 \leq X \leq 100$. If X in a state has value of 50, then some valid operations may not be applicable and so would not appear in the coverage graph. Usually the invariants suffer from incompleteness.

- A bad initialization may stop some operation instances from appearing.

- An operation instance may not occur because we make a finite construction of the coverage graph according to some predefined depth; but, had we continued with graph construction, possibly some more instances might have appeared. But the missing operation instances may be related to the usecases.

3.4.5 Refinement in B

In the B method, the initial specification passes through a succession of refinements and the code can be generated from the final refinement. In practice, the whole refinement sequence is generated less often; one approach is to produce a few refinement steps, generate code and then consistency of the implementation is left to model based testing. Or it could be the case that only one specification is written and code is written manually in relation to this.

Classical B refinement is expressed in terms of a gluing invariant to link the concrete states to abstract states. Further, refinement in B method and various tools supporting B is limited to

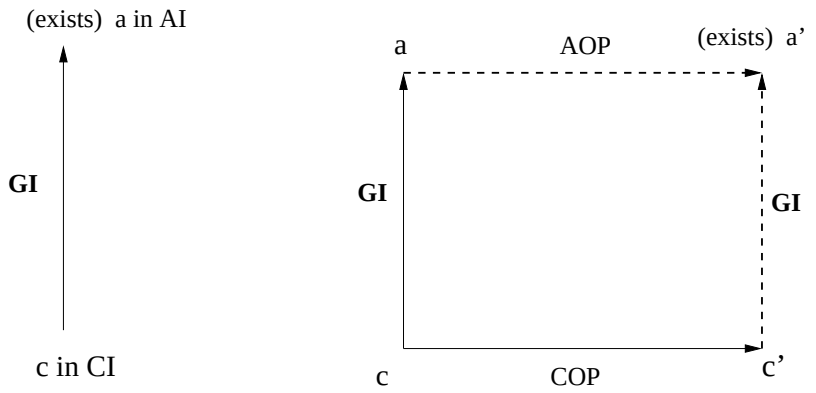


Figure 3.5: Relational Definition of Forward Simulation

forward refinement, or downward simulation [Dun03, LB05]. Let us consider two successive refinements: R_1 and R_2, the latter being a refinement of the former; i.e., R_1 is more abstract in relation to R_2. Let *GI* be the gluing invariant or relation between them. Furthermore, let *AI* and *CI* be the abstract and concrete initial states of the two refinements. Let *AOP* and *COP* stand for an abstract operation and its concrete operations in R_1 and R_2 respectively. Then forward simulation as a relational definition is as in Figure 3.5 [HHS86, LB05]. In other words:

- Every concrete initial state must be related to some initial abstract state
- If concrete state *c* and abstract state *a* are linked by *GI*, and a concrete operation *COP* takes *c* to c', then there must exist an abstract state a' so that the relationship in the diagram holds.

If R_1 and R_2 satisfy these relationships then we can say that R_2 is consistent with R_1. The proof obligations generated by the tools supporting B prove this relationship.

In this background, we will illustrate the problem that we address in our paper. The user scenarios can be described with ease in relation to the initial or the most abstract specification. The initial specification usually has a few high level operations and it would be easy to describe our usecases in terms of these operation sequences, and further the length of these sequences would be small. The description of usecases as a linear sequence of operations will be termed as our initial test cases. However, some usecases may be non-terminating. Consider the use case: `send messages`; this means any number of messages can be sent. This usecase can be expressed by the regular expression: $send.(send)^*$. In such a case, we only take a finite instance of this as our initial test cases, say the sequence: $< send, send, send, send >$.

Usually the initial specification undergoes a succession of refinements, and one such refinement, say R_i, is referred to while writing the code. Let us call this as the *implementation refinement*. Now the problem is that our initial test cases in this situation do not hold any ground because the semantic gap between the code and the initial test cases could be very high. Our approach decreases this gap to a desired level. Our approach upgrades the initial test cases to obtain test cases in sync with the implementation refinement. Refer to Figure 3.6. Let us assume that refinement R_{i+1} has been obtained from R_i through forward simulation. Further R_i satisfies test case T_i; in other words, T_i is a valid trace of R_i. Under this scenario, our approach derives a test case T_{i+1} which is satisfied by R_{i+1}, and in addition T_{i+1} is a trace refinement of T_i.

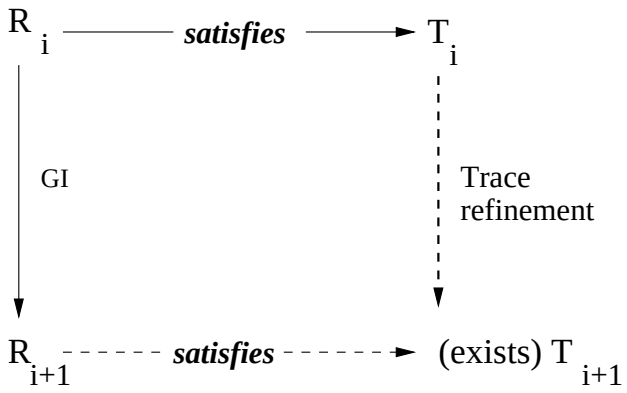


Figure 3.6: A step in the synthesis of Test Cases

3.4.6 The Approach

We refer to the initial specification as R_0. We are given usecase-based test cases for R_0; we have seen that they are easy to construct in relation to the initial specification. Thereafter we repeatedly use the simulation diagram of Figure 3.6 to obtain test cases for successive refinements. This we demonstrate through an example.

3.4.7 An Example

We have taken the leader election problem as our running example. The B machine Leader.mch shows the initial B specification. The Appendix presents this machine along with its two successive refinements. A finite number of processors are arranged in a ring, each processor has a numeric ID. The processor with the highest numeric value is elected as the leader. So, the only usecase would be: *elect a leader*. There is only one state variable and only one operation called *elect* besides the initialization clause. It could be seen that the initial test case in relation to the lone usecase would be: $< init, elect >$.

LeaderR.mch is the first refinement of Leader.mch. It has two new operations, called *accept() and reject()* in addition to the operations of the original machine. LeaderRR.mch is a refinement of LeaderR.mch. It has one more new operation called *send()*. As per the rules of refinement a new event introduced in a refinement must terminate. This is usually ensured by assuming a variant and showing that each invocation of the new operation decreases this variant. We will call the new events in a refinement as τ-operations; they are internal operations when we view them from its parent refinement. With this background, we will now discuss the algorithm to synthesize test cases. In this context, we make the following assumptions.

- we assume a flat B machine; i.e., a machine without any hierarchy.
- Each new event introduced through a refinement has an explicit numeric variant which is decreased after every invocation.

3.4.8 The Algorithm

The Algorithm in Table 3.1 assumes that initial test cases are given, and then it synthesizes test cases for subsequent refinements. We will illustrate our algorithm over the leader election

`Algorithm:` **GenerateTestCases**
Input: A specification (R_0) and K refinements as $R_1, \ldots, R_K$,
and GI_i is the gluing invariant between R_i and R_{i-1}.
Output: $K+1$ test sequences in form of graphs: $G_0, \ldots, G_K$
step 1:
Create $K+1$ nodes $I_0, \ldots, I_K$ as initial nodes of $G_0, \ldots, G_K$.
Let I_i receive the assignments of the Initialization clause in refinement *i*.
step 2
Consider all variants in *K* refinements. Restrict constants and set sizes
so that in the worst scenario, the problem size remains small.
Based on this, instantiate the I_K
step 3:
Project I_k backwards to give full instantiation to $I_0, \ldots I_{K-1}$.
step 4:
Complete a linear path in G_0 to represent a user scenario.
Each node and edge is to receive instantiation as in symbolic execution.
i = 1;
step 5:
Looking at G_{i-1}, construct G_i as follows:
Let G_{i-1} have *t* states (nodes) as $I_{i-1} = A_1, \ldots, A_t$
`for` ($j = 2, \ldots, t$) do
Construct for G_i, nodes B_j, and derive assignments to a subset of
state variables in B_j such that: $B_j \wedge GI_i \Rightarrow A_j$.
Construct a path from B_{j-1} to B_j by repeated execution of
τ-operations of R_i.
Strategy: Whenever data values are needed for operation parameters
select the data values (closed terms) in B_j
`endfor`
step 6:
if ($i = K$) stop
else i = i + 1, Goto step 5.

Table 3.1: Algorithm for graph creation

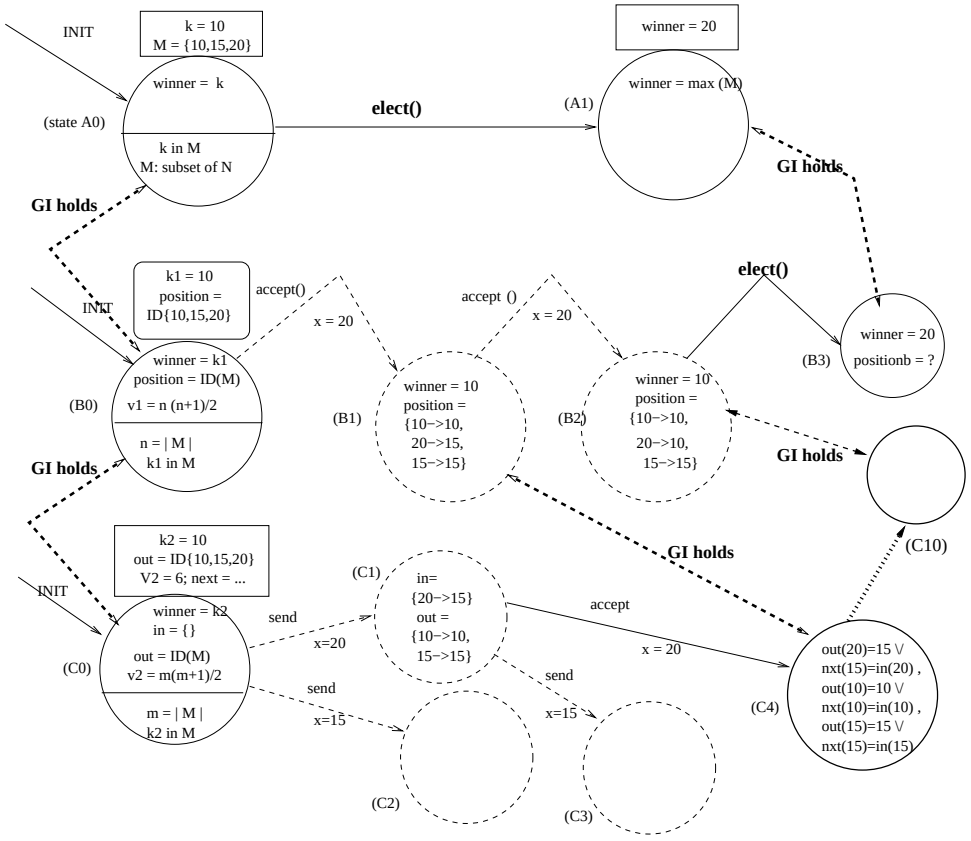


Figure 3.7: Steps in generating the Coverage Graphs

problem as the running example. In the first step, the algorithm creates initial nodes for all the refinements. In Figure 3.7, this is shown by nodes A0, B0, and C0 respectively.

Because of the new events a test case for the current refinement could be larger in size in relation to the corresponding test case of the previous refinement; the extent to which it could be larger is strictly dependent on the variants of the new events in the refinement. As we will soon discuss, the construction algorithm is of exponential complexity. Therefore, we limit the problem size by a suitable instantiation. This task is performed by Step 2 in the algorithm. For the present, the problem size is dependent on variants $v1$ and $v2$ and they should be given small values. For the present and to keep the presentation simple, we give the value of 3 to the size of set M so that $v1$ and $v2$ both get the value of 6. We then can select any values for M, and for the present it is $\{10, 15, 20\}$. And then the instantiation of the constant function $next = \{(10, 20), (20, 15), (15, 10)$ which shows the directions to treat the subset as a ring. This instantiation has been shown in a box on top of node $C0$.

We first give instantiation to the initial node of the last refinement and then we project these instantiations in relation to the gluing invariant backwards so that the initial states of all refinements receive instantiation. In the algorithm, this is done by step 3. In the example, we project the instantiation of $C0$ to $B0$ and $A0$. They have been shown in the boxes attached to the respective nodes.

In step 4 of the algorithm, we symbolically execute the specification (or refinement R_0) in relation to the instantiation and the initial trace. In the process all the nodes in the trace receive instantiation. Note this at node $A1$ in the figure.

Step 5 discusses how we take a fully instantiated trace of a refinement and then derive incrementally a fully instantiated trace of the next refinement. We illustrate this through the example. $< A0, A1 >$ is a fully instantiated path in R_0. Let us assume there exists a state for the first refinement having a gluing relation with $A1$, and let us name it $B3$. Then from the gluing relation and the fact that $A1$ is instantiated, we can give partial instantiation to $B3$; more about this instantiation latter. Since the gluing relation here is ID (identity function), we have for $B3$, $winner = 20$. Next we try to find a path between $B0$ and $B3$ such that the final edge would be labelled with elect() and its prefix would consist of edges all labelled with the new events (τ operations). Our algorithm assumes that such a path exists; we will consider the issue of non-existence later.

The algorithm for finding the shortest path between two such states is NP-Complete [GJ79] because it is a variant of the satisfiability problem. So, we follow a greedy strategy. The strategy states that whenever you need data values for the τ-operations, always select the values from the target state. To be more specific, the data values of the target node has partial instantiation, and some of them may coincide with the data values in the source state. We select values from the state variables which differ from the assignments in the source state.

For the current example, we have `winner`$= 20$ in the target state; so we try to use this value as parameter to the τ-operations. In the process we create the path $< B0, B1, B2, B3 >$. We could have applied the other τ-operation `reject`, but giving it the value of 20 made its precondition false.

Thereafter we repeat step 5 to complete the graph construction for all the refinements. For the current example, we need to obtain a trace for refinement R_2. The gluing relation, say GI_1 between R_1 and R_2 is:

$$position(x) = y \Leftrightarrow out(x) = y \vee next(y) = in(x)$$

We now discover a node to be sync with B1 such that the above gluing relation holds; however, we can discover many nodes denoted by X such that $X \wedge GI \Rightarrow B1$, we discover the most generic one. This means that we assign values to the new node X in a conservative manner. For the present C4 is the state which is in sync with B1 in terms of the gluing relation. Note in the Figure 3.7, how assignment of some state variables of C4 have been given in the form of predicates.

As per our greedy strategy, whenever we select parameters for the τ-operation `send`, we select 15 or 20; this is because these are the places where C0 and C4 differ. In the process we discover the path $< C0, C1, C4 >$. Next, we discover a node $C10$ in sync with node B2. So, now our task is to find a path between C4 till C10 by use of Step 5 once again. And this continues.

3.4.9 Exponential Nature of the Algorithm

Refer to Figure 3.8. The graph at the top shows some traces for refinement R_1 and assume that all the traces shown implement the single trace of R_0, its previous refinement. Let us assume this single trace of R_0 do correspond to one of the usecases. Now as per the rules of refinement, if R_2 implements any of the traces of R_1 then we are done and we can conclude that R_2 implements the original usecase. However, the situation is much more trickier when we have to show that R_2 does not implement the usecase.

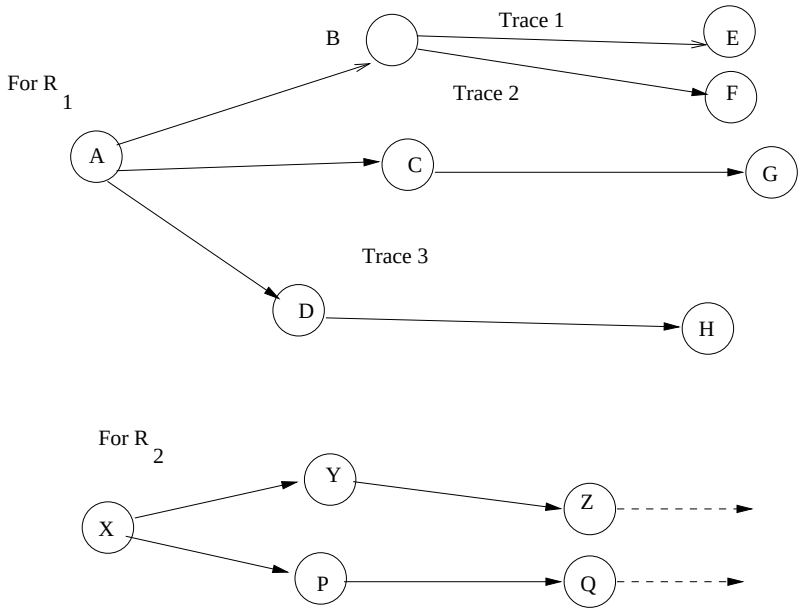


Figure 3.8: Showing non-existence of trace implementation

Let us see how our algorithm works. Consider the situation when the construction for R_1 is over and we are dealing with R_2. The algorithm first takes the trace $< A, B, E >$ and constructs a path $< X, Y, Z, \ldots >$ to show the correspondence. It may so happen that we fail to find such a trace. If so, we next consider the path $< A, B, F >$ and see if a trace for it exists in R_2. If we fail again we try $< A, C, G >$. If we fail for all traces, it may be legal from refinement point of view, but it would also mean that R_2 does not implement the given usecase. And then a warning could be given to the developer to show this deficiency in the refinement process.

Since in the worst case we may have to go for exhaustive enumeration of traces, the algorithm is of exponential nature. However, the length of each trace is limited by the variants of the τ-operations in the refinement. But we always select small values for our variants because of which we do not let the algorithm explode.

Theorem: *The traces obtained by the algorithm in Section 3.4.8 conforms to the commutative diagram of Figure 3.6.*

Proof: Refer to Figure 3.6. The algorithm assumes that R_i, R_{i+1} and T_i are given, and then it computes a trace T_{i+1}. The fact that R_{i+1} satisfies T_{i+1} is obvious, since the trace is one of the behaviours of R_{i+1}. The construction also ensures that T_{i+1} preserves the trace of T_i. More formally, when we treat the new events in R_{i+1} as the internal τ-operations then there exists a *rooted branching bisimulation* [vGW05] between T_i and T_{i+1} with respect to the gluing relation; this simply comes from our construction method. Rooted branching bisimulation preserves trace equivalence under refinement of actions provided actions are atomic [vGW05]. And in B, the operations or actions are atomic. □

3.4.10 Analysis

The following are the highlights of our approach:

- A tool supporting the generation of usecase-based test cases would be semi-automatic in the following sense. The tool would take the initial usecase-based test case instances as input, a specification and a set of successive refinements. The developer may initialize the parameters to limit the solution size. Thereafter, rest of the process could be automated.
- To make the test cases robust one could consider more than one initialization. Furthermore, one could consider multiple instances of the same usecase scenario.
- The approach is capable of generating test cases of shorter length. The current approaches usually create a coverage graph in an ad-hoc manner like: take the initial state and go on applying operation instances till a predefined depth is reached. Our method gives an orientation to the graph creating process; we predefine a depth but our predefined depth has a logical basis.
- Our method can warn the developer of refinement incompleteness in the sense that the refinement omits a certain desired scenario. Thus, our method can help in making the refinements robust.

The following are the low points which needs further research.

- It seems the exponential nature of our algorithm is unavoidable. Even though we limit the problem size, we need intelligent strategies to further cut down the creation of redundant nodes.
- When we show the non-existence of a desired path in a refinement, we need enumeration of all possible paths limited by the variants. Optimization issues in this situation need to be addressed.

3.4.11 Conclusion

We have presented a method in which model based test cases are usecase oriented. Whenever, a specification or a refinement is further refined, our usecase-based test cases can be upgraded to remain in sync with the refinement. Our approach also finds incompleteness in refinements which can be corrected much ahead in the development cycle. Most of the steps in our method can be automated.

Our approach also helps in the formal development process. The method can help in making refinements themselves robust in relation to the original specification. It can also help in cutting down the time to prove proof obligations. After creating a refinement and before proving the proof obligations, the refinement can be tested against the specification in relation to teh test cases derived from our method. If it shows some inconsistencies, then certainly, we have avoided performing some unnecessary proofs.

3.5 Formal View of Developing a Mechanism for Tolerating Transient Faults

The main goal in developing a safety-critical control system is to ensure that the controller is fault-free and that it is able to cope with faults of other system components. The mechanism to

support fault tolerance constitutes a large part of the controller and is often seen as a separate component. We refer to it as *Failure Management System* (FMS). It is a part of the embedded control system as shown in Fig.3.9. The main role of the FMS is to detect erroneous inputs of the system sensors and prevent their propagation into the controller, i.e., provide the controller with the correct information about the system state.

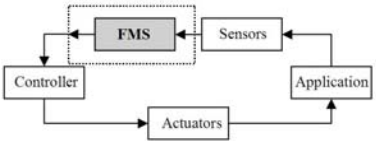


Figure 3.9: Structure of an embedded control system

An acute issue in developing the FMS is design of a mechanism for tolerating and recovering from transient faults of the system components. In [ITLS06] we presented an approach to developing the FMS with the mechanism for tolerating transient faults by stepwise refinement in the B Method.

The formal development of the FMS starts with an abstract specification defining the behaviour of the FMS during one FMS cycle. The stages of such a cycle are:

- obtaining inputs from the environment,
- performing tests on inputs and detecting erroneous inputs,
- deciding upon the input status,
- setting the appropriate remedial actions,
- sending output to the controller either by simple forwarding the obtained input or by calculating the output based on the last good values of inputs,
- freezing the system.

At the end of the operating cycle the system either reaches the terminating (freezing) state or produces a fault-free output. In the latter case, the operating cycle starts again.

Abstract specification. In our abstract specification we model the readings of N multiple homogeneous analogue sensors, measuring the same physical process in the environment. The input values produced by the environment (i.e., sensor data) are assigned non-deterministically.

After obtaining the sensor readings from the environment, the FMS starts error detection. In the abstract specification we model only the result of error detection, which is either TRUE, if an error is detected on the sensor reading on a particular input, or FALSE otherwise.

Based on the results obtained at the detection phase, the FMS non-determinis- tically decides upon the status of an input (i.e., a particular sensor), which may be classified as *fault-free*, *suspected* or *confirmed as failed*. *Suspected* inputs are those faulty inputs which still may recover. The remaining faulty inputs are designated as *confirmed as failed*.

Upon completing analysis, the FMS applies an appropriate remedial action. A *healthy* action is executed if the input is *fault-free*; a *temporary* action if the input is *suspected*, and a *confirmation* action if the input is *confirmed as failed*. While performing a healthy action, the

FMS forwards its input to the system controller. As a result of a temporary action, the FMS calculates the output based on the information about the last good input value. After executing a healthy or a temporary action, the FMS operating cycle starts again. In case of a confirmation action, if the FMS cannot properly function after the input has failed, the system enters the freezing state. Otherwise, it removes the input which has been confirmed as failed from further observations and calculates the output based on the last good input value.

Refining Input Analysis in the FMS. The refinement process of the FMS starts by elaborating on the input analysis procedure. The input analysis is performed gradually by considering inputs one by one until all the inputs are analyzed.

The current value of the input status is calculated based on the results of error detection performed on a certain input and the value of input status obtained at the previous cycle of the FMS. Namely, if the analysed input was previously *fault-free*, it becomes *suspected* after an error is detected. If the input was already *suspected* and an error is detected again, it can either stay *suspected* or become *confirmed as failed*.

A more detailed procedure for determining the input status is based on using a customisable counting mechanism which re-evaluates the status of a particular input at each cycle. It is introduced to distinguish between recoverable and unrecoverable transient faults.

Refining Error Detection in the FMS. Further development of the FMS continues by refining the error detection procedure. The detection mechanism is the most important part of the FMS. The mechanism of error detection relies on a specific architecture of error detection actions called *evaluating tests*. The tests may vary depending on the application domain. For instance, commonly used tests on analogue signals are the magnitude test, the rate test and the predicted values test.

The basic category of evaluating tests is *simple tests*. An input signal may pass through several simple tests, which can be applied in any order. A simple test is executed based solely on the input reading from a sensor. After all simple test associated with a certain input are executed, so called *complex tests* can be performed. The results of the complex tests depend on the results of the simple tests.

Since the system observes homogeneous multiple sensors, for each of N sensor readings the same series of tests can be applied. The tests are executed considering one input at a time, until all the inputs are tested. For each input, we select the tests to be executed according to certain requirements:

req1 each test can be executed at most once on a certain input;

req2 if the test is complex, then all the simple tests it depends on have to be already executed;

req3 if some input has failed, i.e., the error on input is detected, then no more tests on that input should be performed.

The result of the execution of each enabled test is modelled non-deterministically. If the result shows that the test on the input failed, the input is found in error. We should guarantee that for some input to be error free, it should successfully pass all the required tests.

The mechanism of error detection can be further refined. Namely, which tests are enabled for execution depends not only on the requirements listed in req1-3 but also on some additional conditions on the required test frequencies and the internal state of the system:

req4 every test is executed with a certain frequency the test frequency can be different for different tests;

req5 in order for some complex test to be executed, its frequency has to be divisible by the frequencies of all the simple tests required for its execution; This requirement is necessary in order to ensure the application of all required tests on the same data;

req6 the execution of each test may depend on the current internal state of the system.

In order to apply tests according to the given frequencies, we introduce *time scheduling*. There is one global clock guaranteeing that the tests with the same frequency are executed at the same time instances. We model the real time by introducing the event which increments the current time whenever the event is enabled. The progress of time is allowed in two situations:

- after one FMS operation cycle finishes and before the next one starts, or
- when there are no tests enabled for execution under given conditions.

In the latter case, we allow time to progress and possibly the internal system state to be updated until some tests become enabled. After executing all required tests on a particular input, the FMS classifies the input as detected in error or error-free.

The detailed specification of the FMS behaviour while performing the error detection and input analysis is given in the form of the B specification templates. These templates can be instantiated to develop a domain-specific FMS.

As future work we plan to elaborate on creating a methodology for developing the FMS using not only B but UML templates as well.

3.6 Synchronisation-based Decomposition for Event B

In the Event-B, a system is specified as an abstract machine consisting of some state variables and some events (guarded actions) acting on that state. This is essentially the same structure as an action system [BKS83] which describes the behaviour of a parallel reactive system in terms of the guarded actions that can take place during its execution. Techniques for refining the atomicity of operations and for composing systems in parallel have been developed for action systems and such techniques are important for the development of parallel/distributed systems. Different views as to what constitutes the observable behaviour of a system may be taken. In the state-based view, the evolution of the state during execution is observable but not the identity of the operations that cause the state transitions. In the event-based view, the execution of an operation is regarded as an event, but only the the identity of the event is observable and the state is regarded as being internal and not observable. The event-based view corresponds to the way in which system behaviour is modelled in various process algebras such as ACP [BK85], CCS [Mil89] and CSP [Hoa85]. An exact correspondence between action systems and CSP was made by Morgan [Mor90]. Using this correspondence, techniques for event-based refinement and parallel composition of action systems have been developed in [But92, But96]. In this section, we shall use the event-based view of action systems, applying the techniques of [But92, But96] to Event-B machines. For a description of the state-based view of action systems see [BS89].

MACHINE $VM1$

SETS $STATE = \{A, B\}$

VARIABLES n

INVARIANT $n \in STATE$

INITIALISATION $n := A$

EVENTS

$coin \;\widehat{=}\;$ WHEN $n = A$ THEN $n := B$ END

$choc \;\widehat{=}\;$ WHEN $n = B$ THEN $n := A$ END

END

Figure 3.10: Simple vending machine.

3.6.1 Machines as interactive systems

An Event-B machine consists of some state variables, a set of events, each with its own unique name, and an initialisation action. A machine proceeds by firstly executing the initialisation. Then, repeatedly, an enabled event is selected and executed. A system deadlocks if no event is enabled. Fig. 3.10 contains an Event-B system, called *VM1*, specified as a B abstract machine. This is intended to represent a simple vending machine. The state of the machine is represented by the variable n. The machine has two events called *coin* and *choc* respectively. Initially n is set to state A so that only the *coin* event is enabled. When the *coin* event is executed, n is set to B, and only the *choc* event is enabled. Execution of the *choc* event then results in *coin* being enabled again and so on. Thus *VM1* describes a system that alternatively engages in an *coin* event then a *choc* event forever.

As mentioned already, we are taking a purely event-based view of Event-B machines. This means that the environment of a machine only interacts with the machine through its events and has no direct access to a machine's state. The environment of a machine can also control the execution of events by blocking them. This will be seen clearly in Section 3.6.3, where parallel composition of machines is described. Influenced by process algebra, we can view an Event-B machine as an interactive system. As in process algebra, the meaning of two or more such systems interacting will be defined by a parallel composition operator for machines.

Recall that in Event-B an event is specified one of the three following forms:

$$\begin{aligned}
\mathsf{evt} &\;\widehat{=}\; BEGIN\ \ S(v)\ END \\
\mathsf{evt} &\;\widehat{=}\; \text{WHEN } P(v) \text{ THEN } S(v) \text{ END} \\
\mathsf{evt} &\;\widehat{=}\; \text{ANY } x \text{ WHERE } P(x,v) \text{ THEN } S(x,v) \text{ END} \quad ,
\end{aligned}$$

where $P(\ldots)$ is a predicate denoting the guard, x denotes some parameters that are local to the event, and $S(\ldots)$ denotes the action that updates some variables. The variables of the machine

containing the event are denoted by v. Local event parameters x cannot be assigned to, instead their value is constrained by $P(x, v)$. The action part of an event consists of a collection of *assignments* that modify the state simultaneously. An assignments has one of the following three simple forms:

$$x := E(x, v)$$
$$x :\in E(x, v)$$
$$x :| Q(x, v, v')$$

where x are some variables, $E(\ldots)$ denotes an expression, and $Q(\ldots)$ a predicate.

When modelling interactive systems it is convenient to be able to distinguish input and output parameters. This distinction is important when we consider parallel composition later. We adopt the convention that input parameters are denoted by names ending with '?' while output parameters are denoted by names ending with '!'. Influenced by process algebra, we take the view that an event with input parameters models a channel through which a machine is willing to accept input values from the environment whenever that event is enabled. Similary, an event with output parameters models a channel through which a machine is willing to deliver output values whenever that event is enabled.

The machine in Fig. 3.11 models an unordered buffer that is always ready to accept values of type T on the *left* channel, and to output on the *right* channel a value that has been input but not yet output. The order in which values are output is arbitrary. The unordered buffer is modelled by having a *bag* of values as the state variable. A bag is a collection of elements that may have multiple occurrences of any element. We write $bag(T)$ for the set of finite bags of type T. Bags will be enumerated between bag brackets $\prec$ and $\succ$. *Addition* of bags b, c, is written $b + c$, while *subtraction* is written $b - c$. The initialisation statement of *UBuffer1* sets the bag to be empty. The input action *left* accepts input values of type T, adding them to the bag a. Provided a is non-empty, the output action *right* nondeterministically chooses some element from a, removes it from a and outputs it as y.

3.6.2 Refinement and New Events

Recall that new events may be introduced in Event-B refinement, that is, a refined machine may have additional events that have no corresponding events in the abstract machine. New events are required to refine *skip*. This ensures that they have no effect in terms of the abstract state. In order to ensure that the new events do not cause divergence that could prevent progress of the existing events, a refinement should include a variant expression. That variant expression should be decreased by each of the new events.

An example of a refinement with new events is given in Fig. 3.12. *UBuffer2* represents an unordered buffer with an input channel *left* and an output channel *right*. However, instead of having a single bag as its state variable, *UBuffer2* has two bags, b and c. The *left* action places input values in bag b, while the *right* action takes output values from bag c. Values are moved from b to c by the internal action *mid*, which is enabled as long as b is non-empty. The *mid* event is a refinement of *skip* under the gluing invariant $a = b + c$ since the bag sum $b + c$ is unchanged by execution of *mid*. Since b is finite, *mid* will eventually be disabled, so it cannot cause divergence. This is verified by proving that the *mid* event decreases the variant $size(b)$.

MACHINE *UBuffer1*

VARIABLES a

INVARIANT $a \in bag(T)$

INITIALISATION $a := \prec\succ$

EVENTS

left $\widehat{=}$ ANY $x?$ WHERE $x? \in T$ THEN $a := a + \prec x?\succ$ END

right $\widehat{=}$ ANY $y!$ WHERE $y! \in a$ THEN $a := a - \prec y'\succ$ END

END

Figure 3.11: Unordered buffer.

REFINEMENT *UBuffer2*

REFINES *UBuffer1*

VARIABLES b, c

INVARIANT $b \in bag(T) \ \wedge \ c \in bag(T) \ \wedge \ a = b + c$

VARIANT $size(b)$

INITIALISATION $b, c := \prec\succ, \prec\succ$

EVENTS

left $\widehat{=}$ ANY $x?$ WHERE $x \in T$ THEN $b := b + \prec x?\succ$ END

right $\widehat{=}$ ANY $y!$ WHERE $y! \in c$ THEN $c := c - \prec y!\succ$ END

/* NEW EVENT */

mid $\widehat{=}$ ANY z WHERE $z \in b$ THEN $b, c := b - \prec z\succ, c + \prec z\succ$ END

END

Figure 3.12: Unordered buffer with new event.

The new events introduced in a refinement step can be viewed as internal events as found in process algebra. Internal events are not visible to the environment of a system and are

thus outside the control of the environment. In Event-B, requiring a new event to refine *skip* corresponds to the process algebraic principle that the effect of an event is not observable. Any number of executions of an internal action may occur in between each execution of a visible action. If a system reaches a state where internal events can be executed forever, then the system is said to diverge. Requiring new events to decrease a variant ensures that they do not introduce divergence. If machine M is refined by machine N with new events then any observable behaviour of N is also an observable behaviour of M. See Section 3.6.4 provides a more precise definition of what this means.

3.6.3 Parallel Composition

In this section, we describe a parallel composition operator for machines. The parallel composition of machines M and N is written $M \parallel N$. M and N must not have any common state variables. Instead they interact by synchronising over shared events (i.e., events with common names). They may also pass values on synchronisation. We look first at basic parallel composition and later look at parallel composition with value passing.

Basic Parallel Composition of Machines

To achieve the synchronisation effect between machines, shared events from M and N are 'fused' using a parallel operator for events. Assume that m (resp. n) represents the state variables of machine M (resp. N). Variables m and n are disjoint. The parallel operator for events is defined as follows:

$$\begin{array}{l}\text{WHEN } G(m) \text{ THEN } S(m) \text{ END} \quad \parallel \quad \text{WHEN } H(n) \text{ THEN } T(n) \text{ END} \\ \quad \widehat{=} \quad \text{WHEN } G(m) \wedge H(n) \text{ THEN } S(m) \parallel T(n) \text{ END}\end{array}$$

$$\begin{array}{l}\text{ANY } x \text{ WHERE } G(x,m) \text{ THEN } S(x,m) \text{ END} \quad \parallel \\ \text{ANY } y \text{ WHERE } H(y,n) \text{ THEN } T(y,n) \text{ END} \\ \quad \widehat{=} \quad \text{ANY } x,y \text{ WHERE } G(x,m) \wedge H(y,n) \text{ THEN } S(x,m) \parallel T(y,n) \text{ END}\end{array}$$

The parallel operator models simultaneous execution of the events actions and the composite event is enabled exactly when both component events are enabled. This models synchronisation: the composite system engages in a joint event when both systems are willing to engage in that event.

The parallel composition of machines M and N is a machine constructed by fusing shared actions of M and N and leaving independent actions independent. The state variables of the composite system $M \parallel N$ are simply the union of the variables of M and N.

As an illustration of this, consider *N1* and *N2* of Fig. 3.13. *N1* alternates between an a-event and a c-event, while *N2* alternates between an b-event and a c-event. The system $N1 \parallel N2$ is shown in Fig. 3.14. The a-event and b-event of $N1 \parallel N2$ come directly from *N1* and *N2* respectively as they are not joint events (i.e., indedpendent events). The c-event is a joint event and is defined as the fusion of the c-events of *N1* and *N2*. The initialisations of *N1* and *N2* are also combined to form the initialisation of $N1 \parallel N2$. The effect of $N1 \parallel N2$ is that, repeatedly,

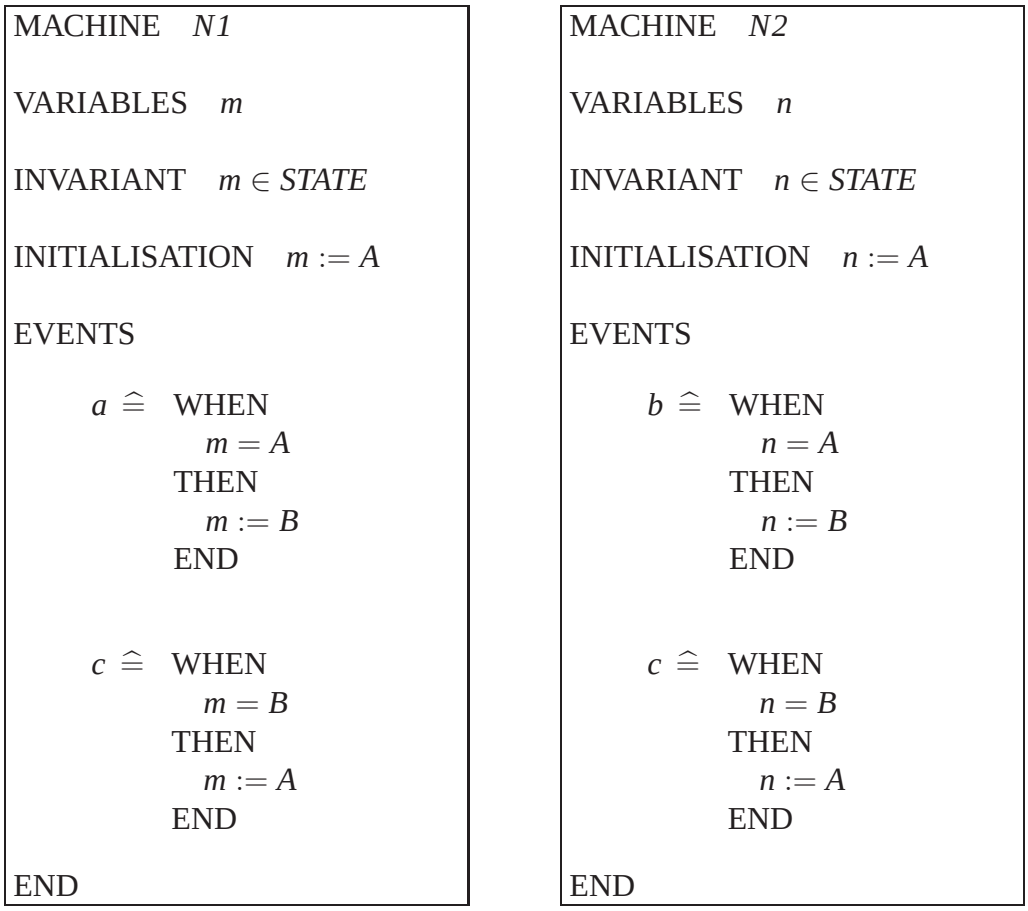


Figure 3.13: Machines with common actions.

the a-event or the b-event can occur in either order, then both systems must synchronise on the c-action.

Parallel Composition with Value-Passing

We extend the parallel operator to deal with input/output parameters and value-passing. An output event from one system is composed with a corresponding input event from another in such a way that the output value from one event becomes the input value for the other event. In order to model the passing of the output value to the input parameter we simply make the corresponding input and output parameters a single joint parameter:

$$\begin{array}{l}\text{ANY } x! \text{ WHERE } G(x!,m) \text{ THEN } S(x?,m) \text{ END} \quad \| \\ \text{ANY } x? \text{ WHERE } H(x?,n) \text{ THEN } T(x!,n) \text{ END} \\ \quad \widehat{=} \quad \text{ANY } x! \text{ WHERE } G(x!,m) \wedge H(x!,n) \text{ THEN } S(x!,m) \parallel T(x!,n) \text{ END}\end{array}$$

Notice that the joint parameter variables $x!$ in the resulting fused event are themselves output parameter. This allows us to fuse input events from further machines with the fused event in a compositional way, thereby modelling broadcast communications.

More generally, fused events may also have additional independent parameters (y and z)

MACHINE $N1 \parallel N2$

VARIABLES m, n

INVARIANT $m \in STATE \ \wedge \ n \in STATE$

INITIALISATION $m, n := A, A$

EVENTS

$a \ \widehat{=}$ WHEN $m = A$ THEN $m := B$ END

$b \ \widehat{=}$ WHEN $n = A$ THEN $n := B$ END

$c \ \widehat{=}$ WHEN $m = B \wedge n = B$ THEN $m := A \parallel n := A$ END

END

Figure 3.14: Composite machine formed through parallel composition.

besides the joint parameters:

$$ev1 \ = \ \text{ANY } x!, y \text{ WHERE } G(x!, y, m) \text{ THEN } S(x!, y, m) \text{ END}$$
$$ev2 \ = \ \text{ANY } x?, z \text{ WHERE } H(x?, z, n) \text{ THEN } T(x?, z, n) \text{ END}$$

$$\begin{array}{lcl} & & \text{ANY } x!, y, z \text{ WHERE} \\ & & \quad G(x!, y, m) \wedge H(x!, z, n) \\ ev1 \parallel ev2 & \widehat{=} & \text{THEN} \\ & & \quad S(x!, y, m) \parallel T(x!, z, n) \\ & & \text{END} \end{array}$$

The fusion of input-input pairs of events is also permitted:

$$ev1 \ = \ \text{ANY } x?, y \text{ WHERE } G(x?, y, m) \text{ THEN } S(x?, y, m) \text{ END}$$
$$ev2 \ = \ \text{ANY } x?, z \text{ WHERE } H(x?, z, n) \text{ THEN } T(x?, z, n) \text{ END}$$

$$\begin{array}{lcl} & & \text{ANY } x?, y, z \text{ WHERE} \\ & & \quad G(x?, y, m) \wedge H(x?, z, n) \\ ev1 \parallel ev2 & \widehat{=} & \text{THEN} \\ & & \quad S(x?, y, m) \parallel T(x?, z, n) \\ & & \text{END} \end{array}$$

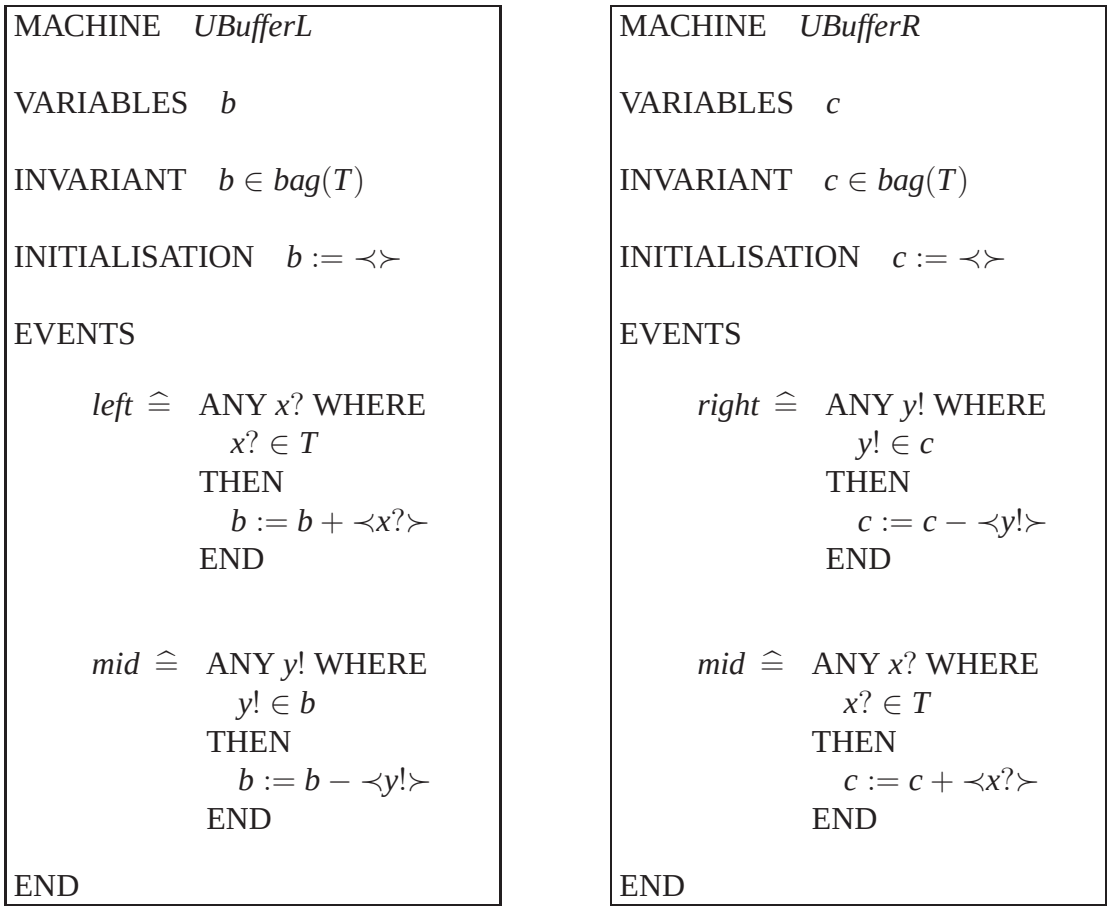


Figure 3.15: Buffers.

The composition of two systems M and N is then constructed by fusing joint input-output pairs of events and input-input pairs of events. As before, independent events remain independent. Fusion of output-output pairs of actions is not permitted. This avoids the introduction of deadlock in situations where two output actions are not willing to output the same value.

Fig. 3.15 presents the machines *UBufferL* and *UBufferR*. *UBufferL* is simply an unbounded buffer with *right* renamed to *mid*, while *UBufferR* has *left* renamed to *mid*. When *UBufferL* and *UBufferR* are placed in parallel, they interact via the *mid* channel, with values being passed from *UBufferL* to *UBufferR*. This can be seen by constructing the composite machine *UBufferL* $\parallel$ *UBufferR* as described above (see Fig. 3.16). The composite machine *UBufferL* $\parallel$ *UBufferR* is the same machine as *UBuffer2* of Fig. 3.12. We have already seen that *UBuffer2* refines *UBuffer1* and thus we have that the simple abstract machine *UBuffer1* consisting of a single variable and two events is refined by the parallel composition of *UBufferL* and *UBufferR*.

3.6.4 CSP Correspondence

In CSP [Hoa85], the behaviour of a process is viewed in terms of the events it can engage in. Value-passing is modelled by grouping events into input channels and output channels. Each process P has an alphabet of events A, and its behaviour is modelled by a set of *failures* F and a set of *divergences* D. A failure is a pair (t, X), where t is a trace of events and X is a set of

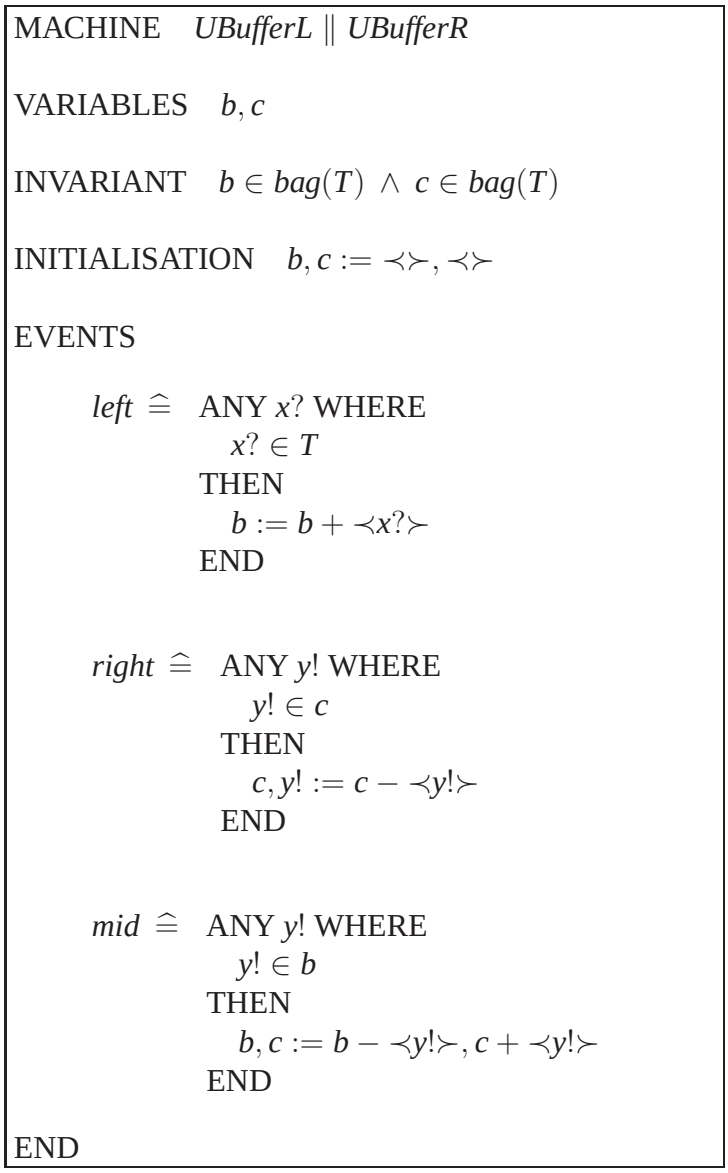


Figure 3.16: Parallel composed buffers.

events; $(t, X) \in F$ means that P may engage in the trace of events t and then refuse all the events in X. A divergence is a trace of events d, and $d \in D$ means that, after engaging the trace d, P may diverge (behave chaotically). Process (A, F, D) is refined by process (A, F', D'), written $(A, F, D) \sqsubseteq (A, F', D')$, if

$$F \supseteq F' \text{ and } D \supseteq D'.$$

In [Mor90], a correspondence between CSP and an event-based view of action systems is described. This involves giving a failures-divergence semantics to action systems, with the execution of actions corresponding to the occurrence of CSP-like communication events. Event-B machines have the same semantic structure and refinement definitions as action systems so these failures-divergence definitions apply equally to Event-B machines [But97]. Let $\{\!|M|\!\}$ represent the failures-divergence semantics of machine M. The definition of $\{\!|M|\!\}$ may be found in [But92, Mor90]. Previously we claimed that if M is refined by N, then any observable

behaviour of N is an observable behavior of M. The observable behaviour of a machine can be represented by its failures-divergence semantics and it can be shown [But92, WM90] that if M is refined by N, then

$$\{\!|M|\!\} \sqsubseteq \{\!|N|\!\}.$$

CSP has both a hiding operator $(P \setminus C)$ for internalising events and a parallel composition operator $(P \parallel Q)$ for composing processes based on shared events. Both operators are defined in terms of failures-divergence semantics: Let $[\![P]\!]$ be the failures-divergence semantics of a process P. Then $[\![P \setminus C]\!]$ is defined by $HIDE([\![P]\!], C)$ and $[\![P \parallel Q]\!]$ is defined by $PAR([\![P]\!], [\![Q]\!])$, where *HIDE* and *PAR* are described in [Hoa85].

We have already seen the synchronised parallel operator for Event-B machines. In order to regard new events in an Event-B refinement as internal, some events of a machine may be flagged as internal. This means that they are treated differently to observable events in the definition of the machines failures-divergence semantics. A hiding operator for maachines $(M \setminus C)$ simply flags events in C as being internal. It can be shown [But92] that the hiding and parallel operators for action systems (i.e., Event-B machines) correspond to the CSP operators; that is, for action systems M and N:

$$\begin{aligned} \{\!|M \setminus C|\!\} &= HIDE(\{\!|M|\!\}, C) \\ \{\!|M \parallel N|\!\} &= PAR(\{\!|M|\!\}, \{\!|N|\!\}). \end{aligned}$$

Significantly *HIDE* and *PAR* are monotonic w.r.t. (failures-divergence) refinement [Hoa85]. Therefore a corollary of the above correspondence is that the hiding and parallel operators for action systems are monotonic w.r.t. (failures-divergence) refinement

3.6.5 Design Technique: Refinement and Decomposition

The derivation of the composite system $UBufferL \parallel UBufferR$ illustrates a design technique that may be used to refine decompose an abstract machine into parallel subsystems: refine the state variables so that they may be partitioned amongst the subsystems, introducing internal events representing interaction between subsystems, then partition the system into subsystems using the parallel operator for machine in reverse. The refinement of the single system can always be performed in a number of steps rather than a single step.

We stated above that synchronised parallel composition of machines is monotonic, i.e., if M is refined by M' and N is refined by N', then $M \parallel N$ is refined by $M' \parallel N'$. This means that when we decompose a system into parallel subsystems, the subsystems may be refined and decomposed independently. This is a major methodological benefit, helping to modularise the design and proof effort.

3.6.6 Concluding

In this section, we have taken an event-based view of machines. An alternative approach is the state-based one. In the state-based view, the refinement rules are the same, but the parallel composition of machines is somewhat different; events are not fused; instead machines share variables through which they interact. The choice between an event-based and a state-based view will depend on the nature of the application being developed. The event-based view is

more suited to the design of message-passing distributed systems, while the state-based view is more suited to the design of parallel algorithms.

These techniques provide a powerful abstraction mechanism since they allow us to abstract away from the distributed architecture of a system and the complex interactions between its subsystems; a system can be specified as a single abstract machine and only in later refinement steps do we need to introduce explicit subsystems and interactions between them. The reasoning required to use these techniques involves standard refinement arguments and variant arguments. The approach is also very modular since the parallel components of a distributed system can be refined and decomposed separately without making any assumptions about the rest of the system. The construction of composite machines modelling synchronised parallel composition is not yet support by the standard B tools nor the RODIN platform. As shown here, the construction is syntactic and thus should easily be supportable by a simple extension/plug-in for the RODIN platform.

3.7 Justifying the soundness of rely/guarantee reasoning

A variety of methods exist for the formal development of sequential programs from specifications. Of special relevance to Rodin are, of course, the methods used on B [Abr96]; however, the ideas used in VDM [Jon90a] are very similar and we expect to be able to migrate concepts from both. More work is required here with respect to Event-B [MAV05].

The methods for sequential programs are compositional in a useful way. In practice, this means that one can use the "operation decomposition" rules in the development process where the programming combinators are applied to specifications rather than programs. It is essential that the specification says all that is needed of an implementation. Technically, one needs to know that the program combinators are "monotonic" in the refinement ordering (i.e. increase in domain, decrease of non-determinacy (while still retaing a sufficient domain)).

It is recognised [Jon03] that it is technically difficult to find compositional methods that cope with concurrency. Jones' old research on rely/guarantee conditions copes with interference in both specifications and program design; the research has been taken much further by a series of PhD students and has been applied in a variety of practical applications. Unfortunately, this left a lack of a clear reference publication on the rely/guarantee method (one would not accuse [dR01] of being either short nor accessible). We have returned to this challenge. Interestingly, we have found a new way of justifying the soundness of rely/guarantee reasoning. We are close to finishing a paper in which we present the formal proofs and expect to cut this down to a journal submission.

In closing, it is worth pointing out that we are not ignoring design of data structures. People working on the Rodin methodology were among the first to recognise that "data reification" (aka "refinement") is often more important than refining programming constructs but we focussed here on language constructs because of the intention to talk about concurrency. But there is an intruiging link! In doing this research and revisiting reasoning about interference, we have found a connection between the choice of data representation and the ability to perform (atomic) updates in the reified state — see [Jon06a].

3.8 Development of distributed transactions in Event-B

Distributed algorithms are difficult to understand and verify. There exists a vast literature on distributed algorithms [Lyn96, SS01] that provides solutions to several problems of distributed systems. Some of the well known problems of distributed systems are distributed mutual exclusion, distributed deadlock detection, global state recording, agreement, consensus, termination detections etc. These algorithms attempt to provide the solution through a well defined mechanism in which various components of the system interact with each other. The solution of many of such problems are based on exchange of messages, tokens or logical clocks. In most cases, these algorithms are either improvement or variant of each other.

There exist several approaches for verification of these algorithms which includes model checking and theorem proving. However, the application of proof based formal methods for systematic design and development of such complex systems is rare. A clear sketch of specification and sound proof of correctness are as important as algorithms itself. The abstraction and refinement are valuable technique for modelling complex distributed systems and reason about them. The important feature of this approach is to formally define an abstract global architecture-independent model of a system and successively refine it to a distributed design in a series of intermediate steps. In principle this allows one to take a top-down approach to development starting with a high level specification and refining this to a distributed implementation.

Formal methods provide a systematic approach to the development of complex systems. Formal methods use mathematical notations to describe and reason about systems. The B Method [Abr96, CM03] is a model oriented state based method for specifying, designing and coding software systems. The B Method provides a state based formal notation based on set theory for writing abstract models of systems. Event B [MAV05] is an event driven approach to system modelling based on B for developing distributed systems. This formal technique consists of the following steps :

- Rigorous description of abstract problem.
- Introduce solutions or details in refinement steps to obtain more concrete specifications.
- Verifying that proposed refinements are valid.

The development methodology supported in B Method is stepwise refinement. This is done by defining an abstract formal specification and successively refining it to an implementable specification through a number of correctness preserving steps. At each refinement step more concrete specifications of a system are obtained. The B Method requires the discharge of proof obligations for *consistency checking* and *refinement checking*. The B Tools Atelier B [Ste97], Click'n'Prove [AC03] provide an environment for generation and discharge of proof obligations required for *consistency checking* and *refinement checking*. Applications of the B method to distributed system may be found in [ACM03, But02, RB05, YB05, YB06]. In this section we present some guidelines to formally develop a model of distributed transaction for replicated database. In the abstract model, an update transactions modifies the abstract database as a single atomic event. In the refinement, update transaction modifies each replica separately.

The remainder of this section is organized as follows: Section 3.8.1 contains background on the problem, Section 3.8.2 Transaction model for abstract database, Section 3.8.3 refinement

with replicated database, Section 3.8.4 presents Gluing Invariants, Section 3.8.5 concludes the material.

3.8.1 Distributed Transactions

A distributed system [SS01] is a collection of autonomous computer systems that cooperate with each other for successful completion of a distributed computation. A distributed computation may require access to resources located at participating sites. A distributed transaction may span several sites reading or updating data objects. A typical distributed transaction contains a sequence of database operations which must be processed at all of the participating sites or none of the sites to maintain the integrity of the database [SKS01]. Assuming that each site maintains a log and a recovery procedure, commit protocols [GR93, SKS01] ensure that all sites abort or commit a transaction unanimously despite multiple failures.

System availability is improved by the replication of data objects in a distributed database system. It is advantageous to replicate data objects when the transaction workload is predominantly read only. However, during updates, the complexity of keeping replicas identical arises due to site failures and conflicting transactions. We consider *read anywhere write everywhere* replica control protocol [ÖV99] for a distributed database system. An update transaction which spans several sites issuing a series of read/write operations is executed in isolation at a given site. The basic idea used in this paper is to allow update transactions to be submitted at any site. This site, called the coordinating site, broadcasts update messages to replicas at participating sites. Upon receipt of update requests, each site starts a sub transaction if it does not conflict with any other *active* transactions at that site. The coordinating site decides to commit if a transaction commits at all participating sites. The coordinating site decide to abort it if it aborts at any participating site.

One of the important issues to be addressed in the design of replica control protocols is consistency. The *One Copy Equivalence* [BHG87, ÖV99] criteria requires that a replicated database is in a mutually consistent state only if all copies of data objects *logically* have the same identical value.

3.8.2 Transaction Model for Abstract Central Database

The abstract model of transactions is given in Fig. 3.17 as a B machine. The abstract model maintains a notion of *central or one copy database*. *TRANSACTION*, *OBJECT* and *SITE* are defined as sets. The *TRANSSTATUS* is an enumerated set containing value *COMMIT,ABORT and PENDING*. These values are used to represent the global status of transaction. The abstract variable *trans* refers to *started* transactions. This variable is defined as $trans \in \mathbb{P}(TRANSACTION)$ as shown in invariant. The variable *trans* is initialized as $trans := \varnothing$. The database is represented by a variable *database* as total function from *OBJECT* to *VALUE*. The variable *database* is initialized non deterministically. A mapping $(o \mapsto v) \in database$ indicate that an object o has value v in database.

The variable *transobject* is total function which maps a transaction to powerset of objects. A mapping $(t_i \mapsto o_i) \in transobject$, where $t_i \in trans$ and $o_i \in \mathbb{P}(OBJECT)$, indicate that transaction t_i either read or write to the data objects in set o_i. The variable *transeffect* is a total function which maps a transaction to object and corresponding value of object. A mapping

MACHINE *Database*
SETS *TRANSACTION; OBJECT; VALUE;*
TRANSSTATUS={COMMIT,ABORT,PENDING}
VARIABLES *trans, transstatus, database, transeffect, transobject*
INVARIANT $trans \in \mathbb{P}(TRANSACTION)$
$\wedge$ $transstatus \in trans \rightarrow TRANSSTATUS$
$\wedge$ $database \in OBJECT \rightarrow VALUE$
$\wedge$ $transeffect \in trans \rightarrow (OBJECT \nrightarrow VALUE)$
$\wedge$ $transobject \in trans \rightarrow \mathbb{P}(OBJECT)$
$\wedge$ $\forall t.(t \in trans \Rightarrow dom(transeffect(t)) \subseteq transobject(t))$
INITIALISATION
$trans := \emptyset$ || $transstatus := \emptyset$
|| $transeffect := \{\}$ || $transobject := \{\}$
|| $database :\in OBJECT \rightarrow VALUE$
OPERATIONS
$StartTran(tt) \cong$; $CommitWriteTran(tt) \cong$;
$val \leftarrow ReadTran(tt,ss) \cong$; $AbortWriteTran(tt) \cong$;

END

Figure 3.17: Abstract Model of Transactions in B

$(t \mapsto (o \mapsto v)) \in transeffect$, where $t \in trans$, $o \in OBJECT$ and $v \in$ VALUE indicate that transaction t will update object o to value v. The invariant $dom(transeffect(t)) \subseteq transobject(t)$ indicate that all objects to be updated must be a part of transaction objects.

The abstract B Machine also contains the parameterized operations *StartTran*, *CommitWriteTran*, *AbortWriteTran* and *ReadTran*. The B specification of these operations are given in Fig. 3.18. The parameterized event *StartTran(tt)* models an event of starting a new transaction *tt*. Guards of event ensure that *tt* is a fresh transaction. The event *CommitWriteTran(tt)* models commit of an update transaction. Guards of this event ensures that when a *pending* transaction *tt* commits, the abstract *database* is modified with the effects of transaction and its status is set to *Commit*. Similarly, the event *AbortWriteTran(tt)* models *abortion* of update transaction. When a *pending* transaction *aborts*, transaction status is set to *Abort* and its effects are not written to database. The event *ReadTran(tt,ss)* models commit of a read-only transaction *tt* at site *ss*. When a *pending* read-only transaction commits, it reads the data objects from abstract database. It can be noticed that in the abstract model that an update transaction commits atomically by updating the database with its effect. Its effects are not written to database in case of aborts.

3.8.3 Refinement with Replicated Database

The Initial part of the refinement is given in Fig. 3.19. In the refinement notion of *replicated database* is introduced. It may be noted that in abstract model given in Fig. 3.18, an update transaction perform updates on abstract central database whereas in a refined model an update transaction updates all copies of replica at various sites separately. Similarly, a read only transaction reads the data from the replica maintained at the site of submission of transaction. The abstract variable *database* is replaced by a variable *replica* in the refinement. The new variables *coordinator*, *replica*, and *freeobject* are introduced in refinement. A mapping of form $(t \mapsto s) \in coordinator$ imply that site s is a coordinator site for transaction t. A coordinator site

for a transaction is determined when a transaction is started. Each site maintains a replica of database. The variable *replica* is initialized non deterministically to have same value of each data object at each site. Subsequently, these objects may be modified by update transactions.

StartTran(tt) $\cong$
  PRE $tt \in TRANSACTION$
  THEN *SELECT* $tt \notin trans$
    THEN $trans := trans \cup \{tt\}$ || $transstatus(tt) := PENDING$
      || *ANY objects, updates*
        WHERE $objects \in \mathbb{P}_1(OBJECT) \wedge updates \in objects \rightarrow VALUE$
        THEN $transobject(tt) := objects$ || $transeffect(tt) := updates$
  END *END* *END;*

CommitWriteTran(tt) $\cong$
  PRE $tt \in TRANSACTION$
  THEN *SELECT* $tt \in trans \wedge transstatus(tt) = PENDING \wedge dom(transeffect(tt)) \neq \emptyset$
    THEN $transstatus(tt) := COMMIT$ || $database := database \lhd transeffect(tt)$
  END *END;*

AbortWriteTran(tt) $\cong$
  PRE $tt \in TRANSACTION$
  THEN *SELECT* $tt \in trans \wedge transstatus(tt) = PENDING \wedge dom(transeffect(tt)) \neq \emptyset$
    THEN $transstatus(tt) := ABORT$
  END *END;*

$val \leftarrow$ *ReadTran(tt,ss)* $\cong$
  PRE $tt \in TRANSACTION \wedge ss \in SITE$
  THEN *SELECT* $tt \in trans \wedge transstatus(tt) = PENDING$
    $\wedge\ dom(transeffect(tt)) = \emptyset$
    THEN $val := transobject(tt) \lhd database$ || $transstatus(tt) := COMMIT$
  END *END;*

Figure 3.18: Operations of Abstract Model

REFINEMENT *Replica*
REFINES *Database*
SETS *SITE, SITETRANSSTATUS={commit,abort,precommit,pending}*
VARIABLES …….
…….
coordinator, sitetransstatus, freeobject, replica
INVARIANT …….
$\wedge$ $coordinator \in trans \rightarrow SITE$
$\wedge$ $sitetransstatus \in trans \rightarrow\!\!\!\!\!\!\rightarrow (SITE \rightarrow\!\!\!\!\!\!\rightarrow SITETRANSSTATUS)$
$\wedge$ $replica \in SITE \rightarrow (OBJECT \rightarrow VALUE)$
$\wedge$ $freeobject \in SITE \leftrightarrow OBJECT$

INITIALISATION …….
……..
|| $coordinator := \emptyset$ || sitetransstatus $:= \emptyset$
|| $freeobject := SITE \times OBJECT$
|| *ANY data WHERE data* $: OBJECT \rightarrow VALUE$
THEN $replica := SITE \times \{data\}$ *END*

Figure 3.19: Initial Part of Refinement

A mapping $(s \mapsto (o \mapsto v)) \in$ replica indicate that site *s* currently has value *v* for object *o*. The variable *freeobject* keeps record of objects at various sites which are *free* i.e. those object on which a *lock* may be acquired.

The variable *sitetransstatus* maintains the status of all started transaction at various sites. A mapping of form $(t \mapsto (s \mapsto \text{commit})) \in$ *sitetransstatus* indicate that *t* has committed at site *s*. The new events such as *IssueWriteTran, BeginSubTran, SiteAbortTx, SiteCommitTx, ExeAbort-Decision* and *ExeCommitDecision* are introduced in operations. The events in the refinement are triggered within the framework of two phase commit protocol [BHG87]. These events are either coordinator site events or participating site events as given in Fig. 3.20 and Fig. 3.21.

StartTran(tt) $\widetilde{=}$ ……

IssueWriteTran(tt) $\widetilde{=}$
SELECT ……..
$\wedge$ *transobject(tt)* $\subseteq$ *freeobject[{coordinator(tt)}]*
$\wedge \forall tz.(tz \in \text{trans} \wedge (coordinator(tt) \mapsto tz) \in activetrans$
$\Rightarrow transobject(tt) \cap transobject(tz) = \varnothing$)
THEN ……..
|| *freeobject := freeobject - {coordinator(tt)}* $\times$ *transobject(tt)*
END;

CommitWriteTran(tt) $\widetilde{=}$
SELECT …….
$\wedge \forall ss.(ss \in SITE \Rightarrow sitetransstatus(tt)(ss) = precommit)$
$\wedge \forall ss, oo \cdot (ss \in \text{SITE} \wedge oo \in OBJECT \wedge oo \in transobject(tt) \Rightarrow (ss \mapsto oo) \notin freeobject)$
THEN ……..
transstatus(tt) := COMMIT
|| *replica(coordinator(tt))* := *replica(coordinator(tt))* $\lhd$ *transeffect(tt)*
|| *freeobject := freeobject* $\cup$ {*coordinator(tt)*}$\times$ *transobject(tt)*
END;

AbortWriteTran(tt) $\widetilde{=}$
SELECT …….
$\wedge \exists ss.$ ($ss \in SITE \wedge sitetransstatus(tt)(ss) = abort$)
THEN ……
transstatus(tt) := ABORT
|| *freeobject := freeobject* $\cup$ {*coordinator(tt)*} $\times$ *transobject(tt)*
END;

val $\leftarrow$ *ReadTran(tt,ss)* $\widetilde{=}$
SELECT …….
$\wedge$ *transobject(tt)* $\in$ *freeobject[{ss}]*
$\wedge$ *ss = coordinator(tt)*
THEN ……..
val := transobject(tt) $\lhd$ *replica(ss)*
|| *transstatus(tt):=COMMIT*
END

Figure 3.20: Coordinator Site Events

Coordinator Site Events

The coordinator site events are given in Fig. 3.20. The submission of a fresh transaction *tt* is modelled by the event *StartTran(tt)*. The site of submission of transaction is assigned as a coordinator site of that transaction. The event *IssueWriteTran(tt)* models the *issuing* of an update transaction at the coordinator. The guard of *IssueWriteTran(tt)* ensures that a transaction *tt* is issued by the coordinator when all active transactions running at the coordinator site of *tt* are not in *conflict* with *tt*.

The *CommitWriteTran(tt)* models the commit event of an update transaction. An update transaction *tt* globally commits only if all participating sites are ready to commit it, i.e., it has status *pre-commit* at all sites. As a consequence of the occurrence of the *commit* event at the coordinator, the replica maintained at the coordinator site is updated with the transaction effects. The data objects held for transaction *tt* are declared *free* and the status of the transaction at the coordinator site is set to *commit*. The *AbortWriteTran(tt)* event ensures that an update will abort if it has aborted at some participating site. A *pending* read-only transaction *tt* returns the value of objects in the set *transobject(tt)* from the replica at its coordinator.

Participating Site Events

The participating site event are given in Fig. 3.21. The *BeginSubTran(tt,ss)* event models starting a subtransaction of *tt* at participating site *ss*. The guard of *BeginSubTran(tt)* ensures that a sub transaction of *tt* is started at participating site *ss* when all transactions *tz* running at *ss* are not in *conflict* with *tt*. A participating site *ss* can independently decide to either pre-commit or abort a subtransaction. The events *SiteCommitTx(tt,ss)* and *SiteAbortTx(tt,ss)*, model pre-committing or aborting a subtransaction of *tt* at *ss*.

The event of *ExeCommitDecision(tt,ss) and ExeAbortDecision(tt,ss)* model commit and abort of *tt* at participating site *ss* once the global abort or commit decision has been taken by the coordinating site. In the case of global commit, each site updates its replica separately.

It can be noticed that in the abstract model, an update transaction modifies the abstract one-copy database through a single atomic event. In the refinement, an update transaction consists of a collection of interleaved events updating each replica separately. This is achieved by replacing abstract variable *database* by the concrete variable *replica*. The transaction mechanism on the replicated database is designed to provide the illusion of atomic update of a one copy database.

3.8.4 Gluing Invariants

The development methodology supported in B Method is stepwise refinement. This is done by specifying an abstract model and successively transform it to more detailed refinements. For verification of distributed algorithms, rigorous description of problem may be given in abstract model and solution may be introduced in detailed specifications in refinement steps. The properties on the system are defined through invariants. In the refinement checking, B tool generate proof obligation which needs to be discharged to show that a refinement is valid. The Invariant which holds on abstract model also holds on the refinement.

BeginSubTran(tt,ss) $\widehat{=}$
SELECT ….….. $\wedge$ *ss* $\neq$ *coordinator(tt)*
$\wedge$ *transobject(tt)* $\subseteq$ *freeobject[{ss}]*
$\wedge$ $\forall$*tz.*(*tz* $\in$ *trans* $\wedge$ (*ss* $\mapsto$*tz*) $\in$ *activetrans*
$\Rightarrow$ *transobject(tt)* $\cap$ *transobject(tz)* = $\varnothing$)
THEN ……...
|| *freeobject* := *freeobject* - {*ss*}$\times$ *transobject(tt)*
END;

SiteCommitTx(tt,ss) $\widehat{=}$
SELECT ……. $\wedge$ *ss* $\neq$ *coordinator(tt)*
THEN *sitetransstatus(tt)(ss)* := *precommit*
END;

SiteAbortTx(tt,ss) $\widehat{=}$
SELECT …….. $\wedge$ *ss* $\neq$ *coordinator(tt)*
THEN ……..|| *freeobject* := *freeobject* $\cup$ {*ss*} $\times$ *transobject(tt)*
END;

ExeAbortDecision(ss,tt) $\widehat{=}$
SELECT ……. $\wedge$ *ss* $\neq$ *coordinator(tt)*
THEN …….. || *freeobject* := *freeobject* $\cup$ {*ss*} $\times$ *transobject(tt)*
END;

ExeCommitDecision(ss,tt) $\widehat{=}$
SELECT ……. $\wedge$ *ss* $\neq$ *coordinator(tt)*
THEN ……. || *replica(ss)* := *replica(ss)* $\triangleleft$ *transeffect(tt)*
|| *freeobject* := *freeobject* $\cup$ {*ss*} $\times$ *transobject(tt)*
END;

Figure 3.21: Participating Site Events

An abstract machine is refined by applying the standard technique of data refinement. If a statement S that acts on variable a, is refined by another statement T that acts on variable b under invariants I then we write $S \sqsubseteq_I T$. The invariant I is called the gluing invariant and it defines the relationships between a and b. Replacing the abstract variable *database* in machine *Database* by concrete variable *replica* in refinement *Replica* results in proof obligations generated by the B tool. Initially, the only proof obligations that can not be proved involve the relationship between variables *database* and *replica*. These proof obligations were associated with the events *ReadTrans* and *CommitWriteTran*. In order to prove these proof obligations we have to add the following invariant.

$$(ss \mapsto oo) \in freeobject \Rightarrow database(oo) = replica(ss)(oo)$$

This invariant means that a free object *oo* at site *ss* represents the value of *oo* in the abstract database. Due to addition of this invariant B Tool further generate proof obligations. These proofs are discharged by adding new invariants. We need to continue to the point when all proofs are discharged.

We observe that at every stage new proof obligations are generated by B Tools due to addition of new state properties as invariants. In this process at every stage we also discover further system properties to be expressed in model as invariants. By discharging the proof obligations we ensure that refinement is a valid refinement of abstract specifications.

3.8.5 Conclusions

In this section we outlined a formal approach to modelling and analyzing distributed system by means of abstraction and refinement. We have presented a case study on distributed transaction mechanism for replicated database. In the abstract model, an update transaction modifies the abstract one copy database through a single atomic event. In the refinement, an update transaction consists of a collection of interleaved events updating each replica separately. The transaction mechanism on the replicated database is designed to provide the illusion of atomic update of a one copy database.

The system devolvement approach considered is based on Event B which facilitates the incremental development of distributed system. The B tool generates the proof obligations for refinement and consistency checking. The majority of proofs were discharged using the automatic prover of the tool, however some complex proof requires use of the interactive prover. These proofs helps to understand the complexity of problem and the correctness of the solutions. Our experience with this case study strengthens our believe that abstraction and refinement are valuable technique for modelling complex distributed system.

3.9 Verification of Coordinated Exception Handling

Usually, a large part of the system code is devoted to error detection and handling [Cri89, WN04]. However, since developers tend to focus on the normal activity of applications and only deal with the code responsible for error detection and handling at the implementation phase, this part of the code is usually the least understood, tested, and documented [Cri89]. In order to achieve the desired levels of reliability, mechanisms for detecting and handling errors should be developed systematically from the early phases of development [RdFC05]. Ideally, the construction of system fault tolerance mechanisms should follow a rigorous or formal development methodology [BFG02].

Error recovery in concurrent and distributed systems is known to be complicated by various factors, such as high cost of reaching an agreement, absence of a global view on the system state, multiple concurrent errors, etc. These systems require special error recovery mechanisms that suit their main characteristics. The Coordinated Atomic (CA) Actions concept [XRR$^+$95] results from combining distributed transactions and atomic actions for the construction of reliable distributed systems that use competitive and cooperative concurrency. Atomic actions control cooperative concurrency and implement coordinated exception handling [CR86] whilst distributed transactions maintain the consistency of the resources shared by competing actions. CA actions function as exception handling contexts for cooperative systems so that any exception raised in an action is handled in a coordinated manner by all action participants.

In order for CA actions to be applicable for the construction of complex, real-world systems with strict dependability requirements, software development based on CA actions has to be supported by rigorous models, techniques, and tools. Several approaches have been proposed for formalizing the CA action concept with the intention either to give a more complete and rigorous description of the concept [VG00] or to verify systems designed using CA actions [XRR$^+$02]. However, an important aspect of CA actions that has not been addressed by existing work is the modelling and verification of the coordinated exception handling. This is surprising, since exception handling complements other techniques for improving reliability,

such as atomic transactions, and promotes the implementation of specialized and sophisticated error recovery measures. Moreover, in distributed applications where a rollback is not possible, such as those that interact with the environment, exception handling may be the only choice available.

In this work, we examine the problem of specifying CA action-based designs in a way that allows us to verify automatically if these designs exhibit certain properties of interest regarding coordinated exception handling. Moreover, since coordinated exception handling is strongly related with action structuring, it is also necessary to model how CA actions are nested and composed [RPZ03] to define multiple exception handling contexts. We present an approach to modeling CA action-based designs that makes it possible to verify these designs automatically using a constraint solver. The main component of the proposed approach is a formal model of CA actions that specifies the structuring of a system in terms of actions, as well as information relative to exception flow amongst these actions. This model can be directly specified using well-known specification languages, like B [Abr96] and Alloy [Jac02], and verified automatically using the tool sets associated with these languages. This work resulted in a paper presented at this year's ACM Symposium on Applied Computing [CFRR06]. A more detailed description of the proposed approach is available as a technical report [CRR05].

In the proposed approach, developers start by performing traditional activities of a software development process, namely, analysis and architectural design of the system, assuming that the system is concurrent and cooperative. At the same time, they define the scenarios in which the system may fail (fault model), what exceptions correspond to each type of error, and where and how the exceptions are handled (exceptional activity). The specification of the system's fault model and exceptional activity can be conducted as prescribed by some works in the literature [RdFC05]. The result of these activities is a CA action-based design of the system that includes a description of the exceptions that can be raised in each CA action and how they are handled. This design is usually described in a modeling language for CA actions (or simply *modeling language*), for example, the Coala [VG00] formal language, or the FTT-UML [GCR04] profile for the UML.

To verify the CA action-based design, it is necessary to translate it to a formal language with adequate support for automated verification (*verification language*). Examples of such languages are B [Abr96] and Alloy [Jac02]. If the modeling language has a well-defined semantics, like Coala, this translation can be completely automated by a tool. The translation can also be automated for informal notations, like UML profiles, but only partially. Usually, some manual intervention is required to resolve ambiguities. Developers used to formal methods can write the system descriptions directly in the verification language. The choice of using one or two specification languages is based solely on usability issues.

The formal specification produced by translating the CA action-based design to the verification language must adhere to a generic CA actions meta-model specifying the elements of CA actions and how they relate (hereafter called *generic CA actions model*). This meta-model supports a static description of the CA action-based design that ignores temporal information. The elements of the generic CA actions model are `ACTION`, `ROLE`, `PARTICIPANT`, and `ROOT_EXCEPTION`. They are the main concepts used in the definition of CA actions. Some of them, like `ACTION` and `ROLE`, include additional information represented through relations and functions. For example, the set of roles of an action is defined by the `ROLES` relation, which associates actions to their respective roles. Both the formal specification and the generic CA actions model are described in the verification language. Up to now, we specified generic

CA actions models using B and Alloy as verification languages [CRR05]. Developers can use either of them to formalize CA action-based designs.

A system is verified by providing its formal specification as input to a constraint solver for the verification language, together with the properties to be verified. The properties of interest that a system must satisfy are split in three categories: basic, desired, and application-specific. Basic properties define the well-formedness rules of the model, the characteristics of valid CA actions. They specify the coordinated exception handling mechanism and how actions are organized. Desired properties are general properties that are usually considered beneficial, although they are not part of the basic mechanism of CA actions. In general, they assume that the basic properties hold. Application-specific properties are rules regarding the flow of exceptions in a specific CA action-based application. Examples of basic (BP) and desired (DP) properties are presented below, stated informally. The paper describing this approach [CFRR06] includes an example of application-specific property.

BP1. If a participant performs a role in a nested action, it must also perform some role in the containing action.

BP2. No cycles in action nesting.

BP3. The exception resolution mechanism of an action resolves all possible combinations of concurrent internal exceptions, unless explicitly stated otherwise.

DP1. Top-level (not nested) CA actions have no external exceptions.

DP2. All internal exceptions of an action are handled in it.

DP3. Any role of an action has handlers for all of the action's internal exceptions, including all resolved ones.

The snippet below presents a formal specification for properties BP1 and DP1 in B, using the ASCII version of the B Abstract Machine Notation [Abr96] supported by B-compliant tools such as ProB.

```
!A.((A:ACTION) =>
  /***** Property BP1  *****/
  !NA.((NA:ACTION & NA:NestedActions[{A}]) =>
    !NAR.((NAR:ROLE & NAR:Roles[{NA}]) =>
      #P.(P:PARTICIPANT & NAR:RolesPlayed[{P}]
      & card(RolesPlayed[{P}] /\ Roles[{A}]) > 0))
  )
  &
  /***** Property DP1 *****/
  (not(#TLA.(TLA:ACTION & A:NestedActions[{TLA}])) => External[{A}] = {})
```

We used the Alloy Analyzer [Jac04] and ProB [LB04] constraint solvers to verify formal specifications in Alloy and B, respectively. If any of the properties of interest does not hold, the constraint solver produces a counterexample. Both constraint solvers, besides generating a counterexample, include a graphic visualizer that provides additional help in the identification of the problem.

The usefulness of the proposed approach was demonstrated by a case study. The target of the case study was the Fault-Tolerant Insulin Pump Therapy [CGP05], a control system with strict reliability requirements for treating patients with diabetes. A detailed description of this

system, including a CA action-based design, is available elsewhere [CGP05]. The proposed approach helped us to uncover some problems in the original, informal design of the system. The first one was an implicit assumption that we discovered while manually translating the UML description of the system to B, according to the proposed approach. The second problem was pointed out by ProB when we tried to verify the formal specification of the system. The problems we found were directly related to the use of coordinated exception handling. In other formal models for specifying CA actions, it would be harder to spot problems like the ones we found because they focus on different aspects of CA action-based systems, such as temporal ordering of events [XRR+02] and dynamic CA action structuring [TLIR04].

3.10 On Specification and Verification of Location-based Fault Tolerant Mobile Systems

This work investigates context aware location-based mobile systems. In particular, we are interested how their behaviour, including fault tolerant aspects, could be captured using a formal semantics, which would then be suitable for analysis and verification. We propose a new formalism and middleware, called CAMA, which provides a rich environment to test our approach. The approach itself aims at giving CAMA a formal concurrency semantics in terms of a suitable process algebra, and then applying efficient model checking techniques to the resulting process expressions in a way which alleviates the state space explosion. The model checking technique adopted in our work is partial order model checking based on Petri net unfoldings, and we use a semantics preserving translation from the process terms used in the modelling of CAMA to a suitable class of high-level Petri nets.

Model-checking carries out the verification of a system using a finite representation of its state space, and exhibits a trade-off between the compactness of this representation and time efficiency. For example, deadlock detection is PSPACE-complete for a compact (bounded) Petri net or equivalent process algebra representation, but polynomial for state graph representation. However, the latter is often exponentially larger, causing the *state space explosion* problem [Val98].

Mobile systems are highly concurrent causing a state space explosion when applying model-checking techniques. One should therefore use approach which alleviates this problem. In our case, we focus on an approach based on partial order semantics of concurrency and the corresponding Petri net unfoldings [McM92]. A finite and complete unfolding prefix of a Petri net is a finite acyclic net which implicitly represents all the reachable states of the original net. Efficient algorithms exist for building such prefixes [Kho03], and complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent systems, because they represent concurrency directly rather than by multidimensional "diamonds" as it is done in state graphs. Referring to the mobility model by CAMA, our approach is particularly suitable for verification of *reachability-like* (or *state*) properties, such as:

- The system never deadlocks, though it may terminate in a pre-defined set of successful termination states.
- Security properties, i.e., all sub-scope participants are participants of the containing scope.

- Proper using of the scoping mechanism, for example: a scope owner does not attempt to leave without removing the scope; agents do not leave or delete a scope when other agents expect some input from the scope; and the owner of a scope does not delete it while there are still active agents in the scope.

- Proper use of cooperative recovery: all scope exceptions must be handled when a scope completes; all scope participants eventually complete exception handling; and no exceptions are raised in a scope after an agent leaves it.

- Application-specific invariants. (Note that the negation of an invariant is a state property, i.e., the invariant holds iff there is no reachable state of the system where it is violated.)

To our knowledge, this is the first attempt to develop unfolding based model checking technique for mobile systems, and consequently it was clear right from the start that we would need to make several decisions of both theoretical and implementation nature.

3.10.1 Our approach

The first problem we had to address was the choice of formal model for capturing the properties and behaviour of mobile systems. A decision was made to focus initially on existing formalisms for mobility with the view that the expected model checkers would then be easily adaptable to the context of CAMA. We decided to investigate two process algebras for mobility, viz. π-calculus [MPW92, Par01] and KLAIM [BBN+03, NFP98]. They represent both synchronous and asynchronous models of distributed systems computation, and so we expected that they would cover a full range of issues relating to the mobility case study. π-calculus allows, in particular, to express dynamic change in the process ability to communicate with the external environment, by passing new *channels* through interactions on previously known channels. It also provides means to express and reason about a variety of security related aspects. The choice of the second model was influenced by our work in other parts of the Ambient Campus case study. KLAIM, in particular, supports explicit localities which are first-class data that can be manipulated by active processes, and coordination primitives for controlled interactions among processes located at network's localities.

We the set out to develop semantics-preserving translations (in the sense of provably generating strongly equivalent labelled transition systems) of both the π-calculus and KLAIM into suitable classes of high-level Petri nets. Our choice for the latter was a modular model of high-level Petri nets, resulting in compositional translation of process expression terms. The work on translating π-calculus has been carried out throughout the duration of RODIN, and resulted in two translations: one for finite π-calculus terms [DKK06b], and recently for general recursive expressions [DKK06c]. The translation of KLAIM was based on some key ideas of our work on the π-calculus, and was first reported in [DKK06a].

3.10.2 Model-checking mobile systems

In [DKK06b] we proposed a translation of a π-calculus term P results in a p-net (a kind of high-level Petri net) with a very close behavioural relationship with the original process expression. More precisely, [DKK06b] showed that the labelled transition system of P is *strongly bisimular* to the labelled transition system of the p-net $PN(P)$.

The theoretical translation described in [DKK06b] is not directly implementable (due to the infinite number of tokens in a special *tag* place employed by the translation), and in our implementation oriented work we set out to modify it in such a way that it could be model-checked by existing tools. Moreover, the specific model-checking technique we decided to use was right from the beginning the unfolding-based verification of Petri nets, which already proved to be efficient in dealing with, e.g., digital asynchronous systems and distributed programs [Kho03].

There are several variations of the unfolding-based model-checking technique, including unfolders and property verifiers. The way p-nets are defined suggests that one should employ a variant which is capable to deal efficiently with coloured tokens and high-level arc annotations. In particular, one should avoid the expansion of the high-level Petri net to its low-level representation and unfolding the latter, since such an approach may yield a huge intermediate low-level net, rendering the whole attempt practically useless (see [KK03] for a thorough discussion of this issue). We therefore decided to use the PUNF model-checker (complemented by the MPSAT property verifier based on a SAT-solver), which directly applies unfolding to high-level nets without expanding them to low-level nets, as described in [KK03]. Having decided on the particular unfolding approach, we then had to address a number of implementation issues, as described in the rest of this paper.

3.10.3 Key implementation issues

A basic problem we faced right from the outset was that p-nets are not compatible with the input required by the unfolder, in that PUNF requires as input a high-level net which: (i) is *strictly* safe in the sense that no place can ever hold more than one coloured token; and (ii) does not include any read-arcs. Another, even more fundamental, issue is that the p-net resulting from the translation will in all but the simplest cases have infinite state space. The reason is that the set of potentially known (or new) channel names is countably infinite, and so any input prefix which receives a name from the environment will necessarily give rise to infinite branching. As a result, a naïve state space exploration through exhaustive enumeration is bound to fail. The way in which addressed these problems is outlined below.

Infinity of new channels. We are primarily interested in checking state properties of mobile systems expressed using π-calculus expressions. For this reason, the main property of channels we are interested in is whether a channel which has been received from the environment is brand new (i.e., fresh) or the same as one of the already known channels. As far as sending to the outside of channels previously restricted is concerned, they are always brand new (fresh), i.e., different from those already known. As the precise identity of a channels which has just become known is irrelevant for our purposes, we proceed as follows: (i) if a restricted channel β is sent outside it becomes known simply as β; and (ii) if input into a channel α happened and the inserted channel is a fresh channel, its identity is set to α, otherwise it is one of the existing known channels. In this way, the number of known channels other than those present initially is bounded by the total number of action names appearing in a π-calculus term. The resulting model can therefore be made bounded and then model-checked. It is worth observing that this treatment of newly known channels is a kind of *symmetry reduction* employed at the level of system modelling.

Read arcs. A specific feature of p-nets is that they use read arcs which test for the presence of tokens without consuming them. We use the standard simulation of a read arc using two directed arcs pointing in the opposite directions. Intuitively, this replaces a non-destructive read operation by a destructive one, followed by a re-write. One can observe that this transformation preserves the interleaving semantics of the net.

Non-safeness. Almost all of the places in p-nets resulting from translation are strictly safe, and so conform to the input required by PUNF. The tag-place, on the other hand, is never safe (actually, in the theoretical translation its marking is always infinite). However, with the decisions about the modelling of known channels made above, we can simulate the working of the tag-place by introducing a *status* places for each action of a process term. The status place holds always one token, 0 or 1 (initially 0), and is suitably updated during the execution of the p-net.

Partial transition expansion. An immediate side-effect of replacing the tag-place by a finite set of status places is that each of the transitions used in the original translation accessing the tag-place needs to be replaced by a set of transition jointly simulating the effect of a single transition in the original translation (note that transition modelling internal communication do not need to be modified). The overall effect was that the number of transition increased but their arc annotations became simpler.

3.10.4 Experimental results

In our experiments described in [KKN06], we used simple 'classroom' scenarios inspired by the Ambient Campus case study. A typical π-calculus-like example has the following form:

$$\textsc{Ness}(n) \stackrel{\text{df}}{=} (\nu h)(\nu h_1)\ldots(\nu h_n)\,(T \mid S_1 \mid \ldots \mid S_n)$$

where T represents a 'teacher' process, and each S_i a 'student' process. Their respective definitions are as follows (note that input prefixes are denoted as $a?b$ and the output ones as $a!b$):

$$\begin{aligned} T &\stackrel{\text{df}}{=} a?\mathit{ness}\,.\,(h_1!\mathit{ness}\,.\,h_1?x_1\,.\,0 \mid \cdots \mid h_n!\mathit{ness}\,.\,h_n?x_n\,.\,0) \\ S_i &\stackrel{\text{df}}{=} h_i?\mathit{addr}_i\,.\,(h!h_i\,.\,h_i!\mathit{done}\,.\,0 \\ &\qquad\qquad +h?\mathit{another}_i\,.\,\mathit{addr}_i!h_i\,.\,\mathit{addr}_i!\mathit{another}_i\,.\,h_i!\mathit{done}\,.\,0) \end{aligned}$$

The idea is that the teacher first receives from the school electronic submission system[2] a channel *ness* using which the students are supposed to submit their work for assessment. The teacher passes this channel to all the students (using n parallel sub-processes), and (also in parallel) then waits for the confirmation that the students have finished working on the assignment before terminating. A student's behaviour is somewhat more complicated. After receiving the *ness* channel, students are supposed to organise themselves to work on the assignment in pairs and, after finishing, exactly one of them sends to the support system (using the previously acquired *ness* channel) two channels which give access to their completed joint work. The students finally notify the teacher about the completion of their work. The property to verify

[2]Called NESS in Newcastle.

is that all the processes involved in the computation successfully terminate by reaching the end of their individual code. For instance, the following move is possible for the initial expression:

$$(\nu h)(\nu h_1)\ldots(\nu h_n)\,(T \mid S_1 \mid \ldots \mid S_n) \xrightarrow{\overline{a}b} (\nu h)(\nu h_1)\ldots(\nu h_n)\,(T' \mid S_1 \mid \ldots \mid S_n)$$

where b is the channel on which links to the completed pieces of coursework are to be submitted, and $T' \stackrel{\text{df}}{=} h_1!b\,.\,h_1?x_1\,.\,0 \mid \cdots \mid h_n!b\,.\,h_n?x_n\,.\,0$.

Examples like that described above allowed us to have easily scalable specifications, which satisfy a correctness property only for some values of n. (Note that for the above example only even n leads to a successful termination). Also, the examples were interesting by exhibiting *different* sources of state space explosion, e.g., coming from parallel composition and choice constructs. The former kind of state space explosion is typically avoided by the unfolding based model-checking techniques, whereas techniques based on interleaving suffer from it. To treat the latter kind of state space explosion we have initiated work on a highly promising novel unfolding-based technique [KKKV05].

Part of the experimental results of [KKN06] are presented in Table 3.2. The meaning of the entries is as follows. The first column identifies a specific instance being model-checked. After that we give the size of the high-level Petri nets derived for the input expression ($\mid P \mid$ and $\mid T \mid$ provide the numbers of places and transitions, respectively), as well as the size of finite prefix of its unfolding ($\mid B \mid$ and $\mid E \mid$ provide the numbers of conditions and events, respectively). The last two columns show the times (in seconds) needed to generate the unfolding using PUNF, and to verify the chosen correctness criterion (i.e., deadlock-freeness) using MPSAT.

Problem	Net		Prefix		Time, [s]	
	$\mid P \mid$	$\mid T \mid$	$\mid B \mid$	$\mid E \mid$	PUNF	MPSAT
NESS(2)	157	200	1413	127	<1	<1
NESS(3)	319	415	5458	366	1	<1
NESS(4)	537	724	24561	1299	6	< 1
NESS(5)	811	1139	93546	4078	46	<1
NESS(6)	1141	1672	281221	10431	411	311
NESS(7)	1527	2335	701898	22662	2904	8

Table 3.2: Experimental results.

After completing the implementation and testing, we compared the performance of our prototype model checking technique with that of a leading verification tool in the area of mobile computing, viz. the Mobility Workbench (MW) [Wor06] (we used its most recent version released in April 2006). The results we obtained indicate in a strong way that our approach performs much better than that offered by the Mobility Workbench. For example, for the series of systems in Table 3.2, MW's performance was comparable for $n \leq 4$, however, for $n = 5$ it crashed with memory overflow after working for over 1 hour. Similar results were obtained for other example systems in [KKN06]. Clearly, comparative testing needs to be extended, but even at this stage we can substantiate our initial belief that partial order reduction should offer significant efficiency gains in the model checking of complex mobile systems.

Further experiments During the series of experiments we conducted, a closer analysis of firing sequences leading to deadlocks detected what might be considered as a 'security breach'. More precisely, the specification allows the whole protocol to terminate successfully in such a way that the students inform the environment (rather than the teacher), and the teacher receives the completion messages from the environment (rather than from the students). Intuitively, this means that the environment acts as an 'intruder in the middle'. Moreover, on the verification side, the possibility of 'too many' communications with the environment should also increase the size of the resulting unfolding. One would therefore expect that the situation becomes different if most of the communication is done within the system's processes. To test this hypothesis, and to address the security issue described above, we re-designed the original specification, in the following way:

$$\textsc{sNess}(n) \stackrel{\text{df}}{=} (\nu h)(\nu h_1)\cdots(\nu h_n)(\nu hres)\,(T'' \mid S_1'' \mid \cdots \mid S_n'')$$

where:

$$\begin{aligned} T'' &\stackrel{\text{df}}{=} a?ness\,.\,(h_1!ness\,.\,h_1!hres.hres?x_1\,.\,0 \\ &\qquad\qquad \mid \cdots \mid h_n!ness\,.\,h_n!hres\,.\,hres?x_n.0) \\ S_i'' &\stackrel{\text{df}}{=} h_i?addr_i\,.\,h_i?report_i\,. \\ &\qquad (h!h_i\,.\,report_i!done\,.\,0 \\ &\qquad\quad + h?another_i\,.\,addr_i!h_i\,.\,addr_i!another_i\,.\,report_i!done\,.\,0) \end{aligned}$$

Intuitively, $\textsc{sNess}(n)$ uses a new secret channel *hres* to ensure that the communication between the students and the teacher about the completion of coursework does not leak outside the system. The results confirmed our initial hypothesis.

Finally, we investigated the effect of reducing the state explosion due the protocol for pairing the students. We therefore model-checked the system assuming that the students know in advance what the pairing is, and who is to communicate with the environment. This, of course, was only possible if n is even. The results indicated that the unfolding copes very well with the state space explosion which is due to concurrency present in the system specification.

3.10.5 Conclusions

The results of this work, reported in [KKN06] as well as in the paper to be published in the REFT book[IKKR05] directly contribute to T2.1, T2.3 and T2.4 of WP2. They indicate that model-checking based on Petri net unfoldings can be a successful technique to deal with distributed systems with mobility. We can also identify at least three challenging areas of future work leading to potentially significant improvements in efficiency and applicability of the present approach:

- To develop an unfolding technique dealing with the read arcs in a direct way, rather than simulating them using pairs of directed arcs. A significant step forward towards such a technique has been made in [KK06], where foundations for the unfolding of inhibitor arcs (which are complementary to read arcs) have been developed.

- To introduce a restricted form of π-calculus recursion (or iteration) still allowing one to use model-checking. This work has already strong theoretical underpinnings contained

in [DKK06c], and we are now working on suitable implementation of a restricted version of the solution presented there.

- To deal with the state space explosion problem caused by aspects other than a high level of concurrency; there are strong indications that the recently proposed *merged processes* [KKKV05] could offer an effective solution.

3.11 Bits 'n Pieces

Datamation used to have a section of short news items: this section collects together notes on some evolving pieces of work.

Design patterns During the second year we have been working on developing a set of design patterns and the decomposition method to be applied in rigorous development of the large scale open agent systems. This work is reported in the second year deliverable D18 (§6) and in more details in [ILRT06]. Four patterns: the specification pattern, the decoupling pattern, the refinement pattern and the decomposition pattern have been proposed and applied in our work on the Ambient Campus Case Study. Application of these patterns allows developers to ensure agent interoperability, decentralised development and code reuse.

Modelling systems based on languages Part of the work being done in the Ambient Campus case study (CS5) involves the use of the Linda and Publish/Subscribe coordination mechanism. However, in the development of our scenario, we have found it necessary to use a languuage which combines the features of both Linda and Publish/Subscribe. As the two systems are not mutually descriptive, it is not possible to properly emulate one in the other.

To this end, a framework was developed around a coordination language which provides the features of both Linda and Publish/Subscribe and this language was defined by way of an Event-B model. However, this resulted in an *ad hoc* semantics that was ambiguous, difficult to understand and difficult to extend safely.

The coordination language has since been redefined in terms of a structural operational semantics (SOS). In doing so, we were forced to confront –and resolve– the ambiguities in the language, including some that we had not been aware of initially.

Despite the different natures and pragmatic intents of Event-B and SOS models, there are interesting correspondences in their notational styles. This correspondence not only makes the task of feeding the work done on the SOS model back into the Event-B model easier, but it also allows a high degree of confidence that the two models represent the same system.

The importance of a clean, unambiguous language semantics cannot be overstated when attempting to build a model of a system based on that language. The correspondence between the two methods suggest there are interesting formal links to be pursued, though even the informal connection we have explored thus far has been very useful. Our work will appear in a forthcoming Technical Report at Newcastle University.

SOS Joey Coleman is presenting a paper at the FM-Ed-06 meeting in Canada.

Relevant publications not cited elsewhere in this report Readers might be interested in [HJR04, JR05, Ran00, JOW06, Jon05a, CJO+05, BGJ06, WV01, Jon81, dRE99, Owi75, OG76, Jon83a, Jon83b, Jon96, Stø90, Col94, Xu92, Bue00, Din00, BS01, dR01, Plo81, Plo04b, Plo04a, Jon96, Bur04, Nip04, CM92].

Chapter 4

The way ahead

This “Month 24” document reports on considerable progress on methodological issues facing Rodin and its eventual users. There still remains much work to be done before we can claim to have a formal development method for fault-tolerant systems.

We see the main challenge for the last year as producing a coherent final document which summarises what has been achieved. This task is made possible because of the significant book being produced by Jean-Raymond Abrial. The WP2 final report will both build on this and confine itself to material which does not fit into the Event-B framework.

We have always made clear that our prime deliverable on the tools front will be coherent support for Event B. We will incorporate as many ideas as possible into this framework. Furthermore, the “plug-in” idea offers us a way of providing support for other concepts. To give just one example, the “UML to B” concept offers a way of seducing a community which is not already using formal methods towards the more formal Rodin tools. There are however other ideas which would require separate support at the formal level. An example here is the use of rely/guarantee style specifications.

We will obviously want to report on other methodoloogies that can help the user develop dependable complex systems. Appendix A sets out a possible structure for the final deliverable on Methodology: we should be delighted to discuss this with our project officer and the chosen reviewers. It is worth stating now that we think it would be counter productive to attempt a “catalogue of everything” in the final report: we will maximise coverage but only in so far as we can offer the reader a coherent story.

Bibliography

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[AC03] Jean-Raymond Abrial and Dominique Cansell. Click'n prove: Interactive proofs within set theory. In *TPHOLs 2003*, pages 1–24, 2003.

[ACM03] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.

[AIR06] B. Arief, A. Iliasov, and A. Romanovsky. On Using the CAMA Framework for Developing Open Mobile Fault Tolerant Agent Systems. University of Newcastle. Technical report, University of Newcastle, 2006.

[BBN+03] L. Bettini, V. Bono, R. De Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice. In *Global Computing*, volume LNCS 2874 of *LNCS*, pages 88–150. Springer-Verlag, 2003.

[BFG02] Cinzia Bernardeschi, Alessandro Fantechi, and Stefania Gnesi. Model checking fault tolerant systems. *Software Testing, Verification, and Reliability*, 12:251–275, December 2002.

[BGJ06] D. Besnard, C. Gacek, and C. B. Jones, editors. *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*. Springer, 2006. ISBN 1-84628-110-5.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BK85] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoret. Comp. Sci.*, 37:77–121, 1985.

[BKP04] J. Burton, M. Koutny, and G. Pappalardo. Implementing Communicating Processes in the Event of Interface Difference. *Fundamenta Informaticae*, 59:1–37, 2004.

[BKPPK02] J. Burton, M. Koutny, G. Pappalardo, and M. Pietkiewicz-Koutny. Compositional Development in the Event of Interface Difference. In P.Ezhilchelvan and

A.Romanovsky, editors, *Concurrency in Dependable Computing*, pages 479–543. Kluwer Academic Publishers, 2002.

[BKS83] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[BLLP04] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: Gsm 11-11 standard case study. *Software Practice and Experience*, 34(10):915–948, 2004.

[BS89] R.J.R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Sci. Comp. Prog.*, 13:133–180, 1989.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems*. Springer-Verlag, 2001.

[Bue00] Martin Buechi. *Safe Language Mechanisms for Modularization and Concurrency*. PhD thesis, Turku, 2000.

[Bur04] J. Burton. *The Theory and Practice of Refinement-After-Hiding*. PhD thesis, University of Newcastle upon Tyne, 2004.

[But92] M.J. Butler. *A CSP Approach To Action Systems*. D.Phil. Thesis, Programming Research Group, Oxford University, 1992. `http://eprints.ecs.soton.ac.uk/974/`.

[But96] M.J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, September 1996.

[But97] M.J. Butler. An Approach to the Design of Distributed Systems with B AMN. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, April 3-4, 1997, Proceedings*, volume 1212 of *Lecture Notes in Computer Science*, pages 223–241. Springer, 1997.

[But02] Michael J. Butler. On the use of data refinement in the development of secure communications systems. *Formal Asp. Comput.*, 14(1):2–34, 2002.

[CFRR06] Fernando Castor Filho, Alexander Romanovsky, and Cecília Mary F. Rubira. Verification of coordinated exception handling. In *Proceedings of the 21st ACM Symposium on Applied Computing*, pages 680–685, Dijon, France, April 2006.

[CGP05] A. Capozucca, Nicolas Guelfi, and Patrizio Pelliccione. The fault-tolerant insulin pump therapy. In *Proceedings of FM'2005 Workshop on Rigorous Engineering of Fault-Tolerant Systems*, pages 33–42, Newcastle upon Tyne, UK, 2005.

[CJ05] Joey W. Coleman and Cliff B. Jones. Examples of how to determine the specifications of control systems. In M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Proceedings of the Workshop on Rigorous Engineering of Fault-Tolerant Systems (REFT 2005)*, number CS-TR-915 in Technical Report Series, pages 65–73. University of Newcastle Upon Tyne, June 2005.

[CJO+05] Joey Coleman, Cliff Jones, Ian Oliver, Alexander Romanovsky, and Elena Troubitsyna. RODIN (rigorous open development environment for complex systems). In *EDCC-5, Budapest, Supplementary Volume*, pages 23–26, April 2005.

[CM92] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, 1992.

[CM03] Dominique Cansell and D Merry. Foundations of the b method. *Computing and Informatics.*, 22(1-31):2–34, 2003.

[Col94] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications – Application to UNITY*. PhD thesis, Louvain-la-Neuve, June 1994.

[Col06] Joey W. Coleman. Determining the specification of a control system: an illustrative example. In M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Proceedings of the Workshop on Rigorous Engineering of Fault-Tolerant Systems (REFT 2005)*, number 4157 in Lecture Notes in Computer Science. Springer-Verlag, 2006.

[CR86] R. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, SE-12(8):811–826, 1986.

[Cri89] Flaviu Cristian. Exception handling. In *Dependability of Resilient Computers*. BSP Professional Books, 1989.

[CRR05] Fernando Castor Filho, Alexander Romanovsky, and Cecília Mary F. Rubira. Verification of coordinated exception handling. Technical Report CS-TR-927, School of Computing Science, University of Newcastle upon Tyne, 2005.

[DB99] J. Derrick and E. Boiten. Testing refinements of state-based formal specifications. *Software Testing, Verification and Reliability*, 9:27–50, 1999.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Formal Method Europe '03*, volume 670 of *LNCS*, pages 268–284. Springer Verlag, 1993.

[Din00] Jürgen Dingel. *Systematic Parallel Programming*. PhD thesis, Carnegie Mellon University, 2000.

[DJK+99] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz. Model based testing in practice. In *International Conference on Software Engineering*, pages 285–294. ACM Press, 1999.

[DKK06a] R. Devillers, H. Klaudel, and M. Koutny. A Petri net semantics of a simple process algebra for mobility. *Electronic Notes in Computer Science - ENTCS*, 1254, 2006.

[DKK06b] R. Devillers, H. Klaudel, and M. Koutny. Petri Net Semantics of the Finite pi-calculus Terms. *Fundamenta Informaticae*, 70:203–226, 2006.

[DKK06c] R. Devillers, H. Klaudel, and M. Koutny. A petri net translation of pi-calculus terms. In *International Colloquium on Theoretical Aspects of Computing (submitted)*. Springer-Verlag, 2006.

[DNFP98] R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[DNLM05] R. De Nicola, D. Latella, and M. Massink. Formal modeling and quantitive analysis of KLAIM-based mobile systems. In *Applied Computing*, pages 428–435, 2005.

[dR01] W. P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[dRE99] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, 1999.

[Dun03] S. Dunne. Introducing backward refinement into b. In *ZB 2003*, volume 2681 of *LNCS*, pages 178–196. Springer Verlag, 2003.

[EB06] Neil Evans and Michael Butler. A proposal for records in Event-B. In *Formal Methods 2006*, 2006.

[GCR04] Nicolas Guelfi, Guillaume Le Cousin, and Benoit Ries. Engineering of dependable complex business processes using uml and coordinated atomic actions. In *Proceedings of International Workshop on Modeling Inter-Organizational Systems*, pages 468–482, 2004.

[GHM97] J.D. Gannon, R.G. Hamlet, and H.D. Mills. Theory of modules. *IEEE Transactions on Software Engineering*, 13(7):820–829, 1997.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[HHS86] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In *ESOP '86*, volume 213 of *LNCS*, pages 187–196. Springer Verlag, 1986.

[HJJ03] Ian Hayes, Michael Jackson, and Cliff Jones. Determining the specification of a control system from that of its environment. In Keijiro Araki, Stefani Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer Verlag, 2003.

[HJR04] Tony Hoare, Cliff Jones, and Brian Randell. Extending the horizons of DSE. In *Grand Challenges*. UKCRC, 2004. pre-publication visible at http://www.nesc.ac.uk/esi/events.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985.

[IKKR05] A. Iliasov, V. Khomenko, M. Koutny, and A. Romanovsky. On Specification and Verification of Location-based Fault Tolerant Mobile Systems. In *REFT 2005 Workshop on Rigorous Engineering of Fault-Tolerant Systems*. Newcastle Upon Tyne, UK (http://rodin.cs.ncl.ac.uk/events.htm), June 2005.

[Ili06] A. Iliasov. Implementation of Cama Middleware. In A.Iliasov B.Arief and A.Romanovsky, editors, *On Using the CAMA Framework for Developing Open Mobile Fault Tolerant Agent Systems. University of Newcastle*. University of Newcastle, 2006.

[ILRT05] A. Iliasov, L. Laibinis, A. Romanovsky, and E. Troubitsyna. Towards formal development of mobile location-based systems. In *REFT'05 – Workshop on Rigorous Engineering of Fault Tolerant Systems*, July 2005.

[ILRT06] A. Iliasov, L. Laibinis, A. Romanovsky, and E. Troubitsyna. Rigorous development of fault tolerant agent systems. Technical Report TR762, Turku Centre for Computer Science, March 2006.

[IR05] A. Iliasov and A. Romanovsky. CAMA: Structured Coordination Space and Exception Propagation Mechanism for Mobile Agents. Presented at ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems: Developing Systems that Handle Exceptions. July 25, 2005. Glasgow, UK, 2005.

[ITLS06] Dubravka Ilic, Elena Troubitsyna, Linas Laibinis, and Colin Snook. Formal development of mechanisms for tolerating transient faults. Technical Report TR763, Turku Centre for Computer Science, April 2006.

[Jac02] D. Jackson. Alloy: A lightweight object modeling notation. *ACM TOSEM*, 11(2), April 2002.

[Jac04] D. Jackson. Alloy home page, 2004.

[JHJ06] C. B. Jones, I. J. Hayes, and M. A. Jackson. Specifying systems that connect to the physical world. Technical Report CS-TR-964, School of Computing Science, University of Newcastle, 2006.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981.

[Jon83a] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon83b] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Jon90a] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon90b] C.B. Jones. *Systematic Software Development using VDM (2nd Edn)*. Prentice Hall, 1990.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Jon03] C. B. Jones. Wanted: a compositional approach to concurrency. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, pages 1–15. Springer Verlag, 2003.

[Jon05a] C. B. Jones. Reasoning about the design of programs. *Royal Soc, Phil Trans R Soc A*, 363(1835):2395–2396, 2005.

[Jon05b] C. B. Jones. Sequencing operations and creating objects. In *Proceedings Tenth IEEE International Conference on Engineering of Complex Computer Systems*, pages 33–36. IEEE Computer Society, 2005.

[Jon05c] Cliff Jones. The case for research into obtaining the right specification, June 2005.

[Jon05d] Cliff Jones. Determining the specification of systems involving humans, July 2005.

[Jon06a] C. B. Jones. An approach to splitting atoms safely. *Electronic Notes in Theoretical Computer Science, MFPS XXI, 21st Annual Conference of Mathematical Foundations of Programming Semantics*, 155:43–60, 2006.

[Jon06b] Cliff B. Jones. Reasoning about partial functions in the formal development of programs. In *Proceedings of AVoCS'05*, volume 145, pages 3–25. Elsevier, Electronic Notes in Theoretical Computer Science, 2006.

[JOW06] Cliff Jones, Peter O'Hearn, and Jim Woodcock. Verified software: a grand challenge. *IEEE Computer*, 39(4):93–95, 2006.

[JR05] Cliff Jones and Brian Randell. Dependable pervasive systems. In *Trust and Crime in Information Societies*, chapter 3, pages 59–90. Edward Elgar, 2005. also visible at http://www.foresight.gov.uk.

[Kho03] V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, School of Computing Science, University of Newcastle upon Tyne, 2003.

[KK03] V. Khomenko and M. Koutny. Branching Processes of High-Level Petri Nets. In *TACAS 2003*, volume 2619 of *Lecture Notes in Computer Science*, pages 458–472, 2003.

[KK06] H.C.M. Kleijn and M. Koutny. Infinite Process Semantics of Inhibitor Nets. In *Petri Nets and Other Models of Concurrency - ICATPN 2006*, volume 4024 of *Lecture Notes in Computer Science*, pages 282–301, 2006.

[KKKV05] V. Khomenko, A. Kondratyev, M. Koutny, and V. Vogler. Merged Processes — a New Condensed Representation of Petri Net Behaviour. In *CONCUR 2005*, volume 3653 of *Lecture Notes in Computer Science*, pages 338–352, 2005.

[KKN06] V. Khomenko, M. Koutny, and A. Niaouris. Applying Petri Net Unfoldings for Verification of Mobile Systems. In *Fourth International Workshop on Modelling of Objects, Components and Agents - MOCA 2006*, pages 161–178. Bericht 272, Department Informatik, Universität Hamburg, 2006.

[KMP97] M. Koutny, L. Mancini, and G. Pappalardo. Two Implementation Relations and the Correctness of Communicated Replicated Processing. *Formal Aspects of Computing*, 9:119–148, 1997.

[KPPK06] M. Koutny, G. Pappalardo, and M. Pietkiewicz-Koutny. Towards an Algebra of Abstractions for Communicating Processes. In *ACSD 2006*. To be published, 2006.

[Lam78] L. Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 2:95–114, 1978.

[LB04] M. Leuschel and Michael J. Butler. ProB: A model checker for B. In *Proceedings of FME'2003*, LNCS 2805, pages 855–874. Springer-Verlag, Pisa, Italy, 2004.

[LB05] M. Leuschel and M. Butler. Automatic refinement checking for b. In *ICFEM '05*, volume 3785 of *LNCS*, pages 345–359. Springer Verlag, 2005.

[LIM+05] Sari Leppänen, Dubravka Ilic, Qaisar Malik, Tarja Systä, and Elena Troubitsyna. Specifying UML Profile for Distributed Communicating Systems and Communication Protocols. Proceedings of Workshop on Consistency in Model Driven Engineering (C@MODE'05), November 2005.

[LTL+05] Linas Laibinis, Elena Troubitsyna, Sari Leppänen, Johan Lilius, and Qaisar Malik. Formal Model-Driven Development of Communicating Systems. Proceedings of 7th International Conference on Formal Engineering Methods (ICFEM'06), LNCS 3785, Springer, November 2005.

[LTL+06] Linas Laibinis, Elena Troubitsyna, Sari Leppänen, Johan Lilius, and Qaisar Malik. Formal service-oriented development of fault tolerant communicating systems. Technical Report TR764, Turku Centre for Computer Science, April 2006.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MAV05] C. Métayer, J.-R. Abrial, and L. Voisin. Event-B language. Deliverable D7, EU-IST "RODIN" Project, 2005.

[McM92] K. L. McMillan. Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. In *CAV 1992*, volume 663 of *Lecture Notes in Computer Science*, pages 164–174, 1992.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mor90] C.C. Morgan. Of wp and CSP. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer–Verlag, 1990.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[NFP98] R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[Nip04] Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a java-like language. Manuscript, Munich, 2004.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[OMG05] OMG. *Unified Modeling Language (UML), Version 2.0*, 2005.

[ÖV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975.

[Par01] J. Parrow. An Introduction to the π-calculus. In Bergstra, Ponse, and Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, University of Aarhus, 1981.

[Plo04a] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, July–December 2004.

[Plo04b] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July–December 2004.

[Ran00] B. Randell. Facing up to faults. *The Computer Journal*, 43(2):95–106, 2000.

[RAO92] D.J. Richardson, A. Leif Aha, and T.O. O'Malley. Specification-based test oracles for reactive systems. In *ICSE '92*, pages 105–118. ACM Press, 1992.

[RB05] Abdolbaghi Rezazadeh and Michael Butler. Some guidelines for formal development of web-based applications in B-Method. In *ZB 05*, pages 472–492, 2005.

[RdFC05] Cecília Mary F. Rubira, Rogério de Lemos, Gisele Ferreira, and Fernando Castor Filho. Exception handling in the development of dependable component-based systems. *Software – Practice and Experience*, 35(5):195–236, March 2005.

[RG01] A. Rensink and R. Gorrieri. Vertical Implementation. *Information and Computation*, 170:95–133, 2001.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[RPZ03] A. Romanovsky, Panos Periorellis, and Avelino Zorzo. Structuring integrated web applications for fault tolerance. In *Proceedings of the 6th IEEE International Symposium on Autonomous Decentralized Systems*, pages 99–106, Pisa, Italy, 2003.

[SKS01] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 4th Edition.* McGraw-Hill Book Company, 2001.

[SLB05] M. Satpathy, M. Leuschel, and M. Butler. Protest: An automatic test environment for b specifications. *Electronics Notes on Theoretical Computer Science*, 111:113–136, 2005.

[Spi88] J.M. Spivey. *Understanding Z.* Cambridge University Press, 1988.

[SS01] Mukesh Singhal and Niranjan G Shivratri. *Advanced Concepts in Operating Systems.* Tata McGraw-Hill Book Company, 2001.

[Ste97] Steria. *Atelier-B User and Reference Manuals*, 1997.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990.

[TLIR04] F. Tartanoglu, N. Levy, V. Issarny, and A. Romanovsky. Using the b method for the formalization of coordinated atomic actions. Technical Report CS-TR: 865, School of Computing Science, University of Newcastle, 2004.

[Val98] A. Valmari. The state explosion problem. In *Advances in Petri Nets*, volume LNCS 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1998.

[VG00] J. Vachon and N. Guelfi. Coala: a design language for reliable distributed system engineering. In *Proceedings of the Workshop on Software Engineering and Petri Nets*, pages 135–154, Aarhus, Denmark, June 2000.

[vGW05] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 2005.

[WM90] J.C.P. Woodcock and C.C. Morgan. Refinement of state-based concurrent systems. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM '90*, volume LNCS 428, pages 340–351. Springer–Verlag, 1990.

[WN04] W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 2004.

[Wor06] Mobility Workbench, 2006. http://www.it.uu.se/research/group/mobility/mwb.

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., 2001.

[XRR+95] Jie Xu, Brian Randell, Alexander B. Romanovsky, Cecília M. F Rubira, Robert J. Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the 25th Symposium on Fault-Tolerant Computing Systems*, pages 499–508, Pasadena, USA, 1995.

[XRR+02] Jie Xu, Brian Randell, Alexander B. Romanovsky, Robert J. Stroud, Avelino F. Zorzo, Ercument Canver, and Friedrich W. von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Transactions on Computers*, 51(2):164–179, February 2002.

[Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming*. PhD thesis, Oxford University, 1992.

[YB05] Divakar Yadav and Michael Butler. Application of Event B to global causal ordering for fault tolerant transactions. In *REFT*, pages 93–103, http://www.eprints.ecs.sotonn.ac.uk/10981/, 2005.

[YB06] Divakar Yadav and Michael Butler. Rigorous design of fault-tolerant transactions for replicated database system using Event B. In *Rigorous development of Complex Fault Tolerant Systems*. Springer (to appear), 2006.

[ZHM97] H. Zhu, P.A.V. Hall, and J.H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

Appendix A

Possible structure of the final WP2 report

- Introduction
  - Systems and their dependability
  - What constitutes a (formal) method?
  - Role of tool support
  - Achieving fault-tolerance
- An *outline* of the Event-B approach to specification and refinement
- Tool support
  - Rodin tool philosophy (database architecture)
  - Proof support
  - Role of abstract model checking
  - Model checking as abstract testing
  - Generating code
- Specifications
  - Using UML and linking to formal specifications (MJB)
  - Model-driven architecture
  - Deriving specifications (CBJ)
  - Coping with evolution
- Extending refinement notions
- Handling mobility