UNIVERSITY OF SOUTHAMPTON

Providing Concurrent Implementations for Event-B Developments

by

A. Edmunds

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Engineering, Science and Mathematics

School of Electronics and Computer Science

March 2010

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by A. Edmunds

The Event-B method is a formal approach to modelling systems which incorporates the notion of refinement. This work bridges the abstraction gap between the lowest level of Event-B refinement and a working implementation. We focus on the link between Event-B and concurrent, object-oriented implementations and introduce an intermediate, object-oriented style specification notation called Object-oriented Concurrent-B (OCB). The OCB level of abstraction hides implementation details of locking and blocking, and provides the developer with a clear view of atomicity using labelled atomic clauses. OCB non-atomic clauses are given Event-B semantics, and OCB atomic clauses map to atomic events. Automatic translation of an OCB specification gives rise to an Event-B model and Java source code. The Java program will have atomicity that corresponds to the formal model (and therefore OCB clauses), and structure that is derived from the OCB model.

We introduce process and monitor classes. Process classes allow specification of interleaving behaviour using non-atomic constructs, where atomic regions are defined by labelled atomic clauses. Monitor classes may be shared between the processes and provide mutually exclusive access to the shared data using atomic procedure calls. Labelled atomic clauses map to events guarded by a program counter derived from the label. This allows us to model the ordered execution of the implementation. The approach can be applied to object-oriented systems in general, but we choose Java as a target for working programs. Java's built-in synchronisation mechanism is used to provide mutually exclusive access to data. We discuss some problems related to Java programming, with regard to locking and concurrency, and their effect on OCB.

The OCB syntax and mappings to Event-B and Java are defined, details of tool support and case studies follow. An extension to OCB is described in which a number of objects can be updated within a single atomic clause; facilitated by Java SDK 5.0 features. The extension allows direct access to variables of a monitor using dot notation, and multiple procedure calls in a clause. We also introduce new features to atomic actions such as a sequential operator, and atomic branching and looping.

Contents

List of Figures

List of Tables

Acknowledgements

Thanks to all who have provided support and guidance during the past five years. In particular, I am very much indebted to Michael Butler for his insight and advice during this undertaking. Many thanks also go to Abdolbaghi Rezazadeh, and Divakar Singh Yadav whose friendship and advice I value tremendously. Additionally I would like to thank my children, who have assisted me greatly, especially in the final year of my work. Thanks also to EPSRC for funding in the early years of my research, and latterly to the HEALF whose assistance made a great difference to my family's situation.

To Alex, Nick and Steph.

Chapter 1

Introduction

The dependability requirements for software vary depending on the effects of its failure. For the most safety critical, and business critical domains there is a need for a very high level of dependability. One approach used to improve dependability is the use of formal methods as part of the development process. Formal techniques have been maturing for decades, and as computing technology advances problem domains create new challenges; at the same time the technology available to address those challenges changes, and new theories evolve to describe the problems formally. The work presented in this thesis forms part of the ongoing research that aims to discover better ways of understanding and specifying software systems, with the aim that they be reliable in use.

1.1 Our Contribution

Our work focusses on code generation for Event-B developments [6, 7, 8, 124]. We bridge the abstraction gap between formal development, and implementation in a object-oriented language with concurrent processes. There are some existing Event-B developments that may make use of concurrency such as the Mondex electronic purse [34], distributed database transactions [161], and an ongoing extension of the work introduced in [48], applied to a Flash File System. However, to date there has been no automated code generation applicable to Event-B development, and we specifically wish to specify developments incorporating concurrency. Code generation does exist for its predecessor classical-B [5], via the implementation level notation B0 [43, 44]; but B0 is not aimed at specifying developments with concurrent processing.

In our work we consider it useful to make use of a modern object-oriented language such as Java [67, 68] as our target language. Java supports concurrency and is a widely used platform. It is a strongly typed language with good support for structured data and encapsulation in the form of classes. Whilst we have chosen Java as a target we would

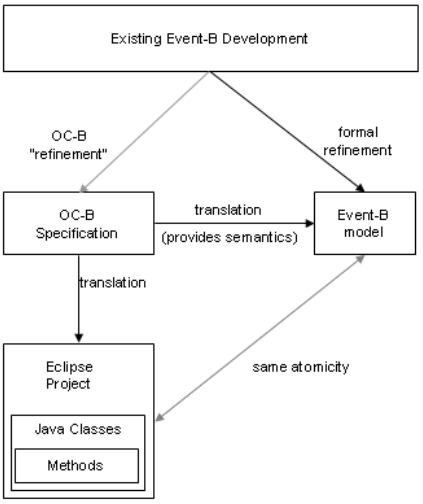


FIGURE 1.1: Extending an Event-B Development with an OCB Specification to Provide an Implementation

reasonably expect to be able to apply the principles of the approach to other similar target languages.

To bridge the abstraction gap between Event-B and the target platform we introduce a notation that we call Object-Oriented Concurrent-B (OCB), see Figure 1.1. OCB was developed to enable the specification of some key features of a concurrent object-oriented development that are not so readily expressible in Event-B. In OCB we make use of the notion of classes for encapsulating data; and procedure calls for accessing the protected data, and constructing new instances. Consider an Event-B development that has proceeded to the point where an object-oriented implementation is desired. At this level a developer will wish to consider details such as processes and their interleaving, and the sharing of data. The OCB notation facilitates the specification of these aspects as well as providing an object-oriented notation which eases the transition to object-oriented implementation. One important aspect of the Event-B approach is that any enabled event may occur, but only one of the enabled events may occur at any given time. When considering a specification involving processes with interleaved executions we seek to introduce the notion of interleaving operations. Event-B contains no facility of this kind since events are atomic, and we additionally need to impose ordering on the executions of an interleaving operation. It is with this in mind that we introduce non-atomic operations, and a sequence operator to specify the points at which interleaving may take place. An OCB specification also embodies some degree of abstraction since synchronization details are hidden from the developer, this will simplify the developer's task of reasoning about the effects of the interactions between shared objects. The approach incorporates aspects of UML-B [89, 110, 137, 138, 140, 142] and builds upon them to facilitate the rigorous specification of concurrent systems. The translation of an OCB specification results in source code and an Event-B model describing the execution of the code. The target language that we choose for this thesis is Java; however the

approach is also applicable to other target languages such as Ada or C. The Event-B model can be shown to refine an abstract model and therefore be considered part of the formal development. The Java correspondence with the formal model is that the formal model represents the implementation. We have not formulated a formal correspondence between the two; but the simplicity of the mapping between the OCB constructs and the formal model, and the simplicity of the mapping between the OCB constructs and Java, leads us to have high confidence in the correspondence.

Event-B is based on the notion of discrete events that occur in systems; an event is represented as an artefact in the formal model, which is also known as an event. In our work when we refer to an event it is clear from the context whether we are referring to the formal artefact rather than the more general meaning: observable events. When addressing issues of concurrency we use the fact that an event is atomic and gives rise to clearly bounded atomic regions in the implementation. We use the notion of labelled atomic clauses; each atomic clause maps to an event. In the mapping to Java we can use synchronized methods and blocks, together with the application of some simple rules, to ensure a representative implementation of the model is produced. In an extension to the approach based on synchronization, we extend the notion of labelled atomic clauses to allow exclusive access to a number of shared objects. To do this we employ locking constructs introduced in a later release of the Java platform. In addition to this we move towards a more implementation oriented notation; for instance we replace parallel constructs in actions with sequential constructs.

In the formalisation of the link between OCB and Event-B we use a textual version of Event-B, similar to that used in [31], and a textual version of our OCB notation for specification. We support the approach using a prototype tool with a translation to Event-B models compatible with the RODIN tool [153], and Java source code. OCB models will be constructed using a prototype GUI based editor rather than a text based specification; so with regard to the tool support neither OCB nor Event-B is currently text based. In the prototype tool we provide a translation to textual style OCB for convenience.

1.2 An Overview of the Thesis

In this chapter we began with a brief introduction to the thesis and describe its contents. In Chapter 2 we provide the context for our work by describing some important concepts which lead to the state of the art in our field, and on which our current work is grounded. The work presented in this thesis draws both on the world of formal methods, and that of programming languages. Formal methods use mathematical notations to describe systems and prove properties about them; and programming languages are translated to instructions that computers use to perform tasks. To assist with our understanding of

current formal approaches we describe some relevant theories, and note their historical significance, leading us to the state of the art in our field. We describe some of the uses of formal methods, and how object-oriented technology has been used in formal methods. We then discuss the technologies that underpin the main focus of our work; we describe the Event-B method, and preliminary work on object-oriented modelling that is part of the UML-B approach. We then briefly describe the implementation language of classical B, B0. Chapter 2 continues with an exploration of the issues involved with Java programming, paying particular attention to problems that arise when specifying concurrent, shared memory implementations using the Java Language Specification 2. We conclude our background with a detailed discussion about how formal, and semi-formal, methods have been brought to bear on the problems faced by developers using Java. Chapter 3 introduces the OCB language; we describe the underlying semantics of interleaving process operations, and shared monitors with atomic procedures. We define the semantics in terms of the Guarded Command Language, and then map the constructs to Event-B. Chapter 4 continues the introduction of the OCB language by introducing object-oriented constructs and their mapping to Event-B, then we present the definition of the mapping of textual OCB to Java code. Chapter 5 gives details of the implementation of the OCB modelling and translation tools, and the integration with the Eclipse Platform and the RODIN tool. We also provide details of how the translator code can be annotated to link the OCB mapping rules to the Java code that implements the mapping rules. In Chapters 6 and 7 we present case studies which describe how OCB may be used to specify object-oriented, concurrent implementations that form part of an Event-B development. Chapter 6 describes the development of a shared buffer with reading and writing processes. An abstract model is used to specify coarse grained atomic events. In subsequent refinements the atomic events are split into a number of atomic actions which refine a single atomic event. We use diagrams based on Jackson Structure Diagrams to visualise the relationships between events of the abstraction and the refinements. We then provide implementation details in an OCB specification and translate the OCB model to Event-B and Java. In Chapter 7 we present a case study presenting a limited number of features of a Flash File System development. We specify the top two layers (the User Application Layer and File System API Layer) of a hierarchical system specification. This chapter serves to highlight some of the shortcomings of the approach and leads into the next chapter, Chapter 8, where we describe how to overcome a number of the restrictions. In Chapter 8 we describe how we can translate to a later version of Java, that of JDK 1.5 and Java Language Specification 3 which is, in turn, reflected in a revised OCB notation. The use of this later Java version facilitates a more flexible mutual exclusion policy, and therefore allows more complex specifications which we describe as transactional clauses. We revise the OCB notation to accommodate the transactional style of specification, and describe the new mapping to Event-B and Java code. Chapter 9 concludes the discussion with an appraisal and suggestions for future work.

Chapter 2

Background

In order to provide a context for our main contributions we discuss some founding formal theories that underpin the state of the art in our field of interest; and give details of various approaches that are applied to system development. Primarily we aim to improve the reliability of software systems using formal methods and, in particular, we are interested in linking a formal development approach with object-oriented implementations incorporating concurrency. So we provide an overview of some fundamental object oriented concepts, and follow this with a general discussion about the influence of object-oriented technology on some formal methods. We follow this by an overview of some approaches that are used to improve the dependability of Java, discussing some formal, semi-formal and non-formal approaches. We then describe the Event-B method [6, 7, 8, 124], which we use to formally model systems. Since we have chosen Java [67, 68] as a platform for our implementations we introduce related programming issues; such as programming constructs, synchronisation, and conditional waiting. Importantly we also highlight some of the problems that may arise when using Java implementations that make use of concurrency constructs. Some of the modelling techniques underlying our approach are based on the UML-B approach [89, 110, 137, 138, 140, 142] - the graphical 'front-end' of Event-B. We therefore give an overview of UML-B which includes details of how classes are modelled and instantiated. There is then a brief overview of classical-B's implementation level notation, B0 [43, 44]. The final section gives more details of the guarded command language which we use in the definition of the OCB language. We then give an overview of our contribution, and provide some general details about our approach.

2.1 The Basis for Formal Methods

Researchers in the field have presented many approaches to formal specification of software systems over many years; many are traceable back to a few fundamental theories. To begin our overview of the formal methods field we discuss some of the fundamental

theories that have formed the basis of research over the years; and due to the sheer volume of work in this field we limit our discussion to what we consider to be the most relevant to this thesis.

2.1.1 Hoare Logic

Hoare logic [76] was first presented in 1969. It provides axioms and inference rules for proving properties of programs. A triple $P \{| S |\} Q$ denotes a precondition P, program, S and post-condition Q. The logic is used to verify that if P holds before an execution of program S then Q will hold after the execution provided it terminates successfully.

The work goes on to present rules for program elements such as assignment, composition, and iteration. An example of the rules is that of iteration,

$$\frac{P \wedge B\{S\}P}{P\{while\ B\ do\ S\}\neg B \wedge P}$$

This states if the assertion P is initially true and the triple is true (the body re-establishes P when B holds) then the equivalent program with P initially true and a while loop with condition B, will eventually establish $\neg B \wedge P$ on termination. However the logic assumes only partial correctness since it does not include the notion of program termination, and simply makes it an assumption in the interpretation.

In [40] Hoare triples are used to verify correctness of programs using inference rules, for instance, for a statement involving a sequential program statement, $P \{| S_1 ; S_2 |\} Q$, the following rule is used,

$$\frac{P \{| S_1 |\} Q \qquad P \{| S_2 |\} Q}{P \{| S_1 ; S_2 |\} Q}$$

and the rule for a simple assignment is written as follows,

$$\frac{P \subseteq Q[x := e]}{P \{|x := e|\}Q}$$

Hoare logic was very influential in its contribution to the state of the art and influences can be seen in the preconditions and postconditions of Design by Contract approaches of Eiffel [111, 112], SPARKAda [2], and JML [28, 102]. Another influence of Hoare logic was in Dijkstra's wp-calculus [51, 52] which later contributes to the semantic definition of the B-method; which is a predecessor of our formal method of interest, Event-B [6, 7, 8, 124].

2.1.2 Guarded Commands

We now move on to look at the Guarded Command Language, which we use later in our work to specify our approach. It was also an influential contribution and the concepts were later extended in Back's work on the refinement calculus [18, 19]. Dijkstra proposed in the 1970's [52, 51, 53] that a formal approach should be used as part of program development, and developed the Guarded Command Language. In our work on OCB we find it useful to describe the behaviour of OCB actions in terms of Dijkstra's Guarded Command Language. Guarded commands are introduced which have the form $G \rightarrow S$ where G is a boolean expression and S is a list of statements. If the guard part of the guarded command is true then the statements in the statement list are applied, updating the state as determined in the statements. The statement may consist of assignments, repeating (looping), or alternative (branching) constructs.

We begin with the assignment statement, where $x := y$ means that after the computation x has the value y. Several statements in the list S can be connected using the semi-colon operator, which is used to indicate sequential computation. In order to describe branching behaviour the language provides the *alternative* construct, where guarded statements are mutually separated by the $\Box$ separator and contained between the pair $if\ fi$. Any statement with a true guard may be non-deterministically selected for evaluation, if no guards are true then the statement is equivalent to abort. The syntax of the *alternative* construct follows,

$$if\ G_0 \rightarrow S_0 \Box G_1 \rightarrow S_1 \Box G_n \rightarrow S_n\ fi$$

The repeating construct is contained between the pair *do od*. While any of the guards are true a statement with a true guard is selected and evaluated, it is typically interpreted that the loop only terminates when all the guards are false. The terminating behaviour of the loop do to the false guard is considered to represent normal termination of a program. However if it is the case that all branches are false in the alternative construct then this is considered to be abnormal termination, that is the program aborts.

$$do\ G_0 \rightarrow S_0 \Box G_1 \rightarrow S_1 \Box G_n \rightarrow S_n\ od$$

In this work Dijkstra also introduce the concept of *weakest preconditions*, using the notation $wp(S, R)$ where S is a list of statements and R is a condition on the state. The weakest precondition is used to identify the set of all initial states such that when the statements S are applied the program terminates and postcondition R holds. wp is a predicate transformer that relates a precondition to any post-condition R.

2.1.3 The Z-notation

The Z-notation [143] was developed in the 1970's. An early version of the Z notation was described in a paper by Abrial, Schuman and Meyer [11].

Z is formally underpinned by set-theory and first-order predicate logic; the developer describes the system being modelled using set-theoretic constructs and predicates. In order to make the specification activity easier Z incorporates the notion of structuring, using schemas. A schema can be used to specify the initial state of the system, and the transformations to successor states are described using operation schemas. Operations are atomic, and changes to state are described in terms of before and after states. Invariant properties are used to describe constraints on the state of the system, i.e. the values that the variables can take.

The format of a schema follows; with schema name n, declarations D and predicates P,

$$\begin{array}{|l} \multicolumn{1}{c}{n} \\ \hline D \\ \hline P \\ \hline \end{array}$$

the declaration part is used to introduce and type variables and import existing schemas. The properties part applies some constraints to the variables introduced in the declaration part. An example of its use, taken from Spivey's tutorial [144], is the simple property that follows,

$$\begin{array}{|l} \multicolumn{1}{c}{Aleph} \\ \hline x, y : \mathbb{Z} \\ \hline x < y \\ \hline \end{array}$$

which states that x and y are integers and that $x < y$ must hold.

Schemas can be combined to form a new schema, and thus the predicate parts may refer to variables imported from other schemas, as well as variables that are global (not discussed here). The following operation declaration uses *Aleph* and the notation Ξ indicates that the state may not change during the operation, alternatively Δ could be used to indicate that the variables of the imported schema may change.

$$\begin{array}{|l} \multicolumn{1}{c}{GetVal} \\ \hline \Xi Aleph \\ out! : \mathbb{Z} \\ \hline out! = x \\ \hline \end{array}$$

Refinement is used in the development process to proceed toward a concrete implementation. Abstract data types are data refined to concrete data types; and explicit control variables can be added to control execution in operation refinement. In data refinement a schema is produced, where the predicate part relates the variables of the abstract with the variables of the refinement.

2.1.4 Refinement Calculus

The origins of the refinement calculus can be traced to works of preceding sections involving Dijkstra [50] and the Z-notation of [143], and additionally Morgan's work on programming from specifications [118]. Back introduced the refinement calculus in [18, 19], presenting a formal system which can used to show proof of refinement at each stage of a step-wise refinement. Later work [21] introduced the notion of contracts. Contracts are held between agents and regulate the behaviour of agents. A contract is described using the notation $\sigma\{|S|\}q$ which states that an agent can satisfy the contract if from an initial state σ it can establish that contract S satisfies the postcondition q. It can achieve this either by not breaching the contract; or it is released from the contract by an assumption that is violated.

Contracts can be refined by other contracts, if a contract S_1 is refined by a contract S_2 then we write $S_1 \sqsubseteq S_2$. In this case, for any initial state σ and postcondition p, if $\sigma\{|S_1|\}p$ holds then $\sigma\{|S_2|\}p$ also holds. Informally this means that any condition established by the abstract specification can be established by the refinement.

2.1.5 Classical B

The B Method [5] developed by J.R Abrial is a set-theoretic modelling method, mathematical theory and notation. The theory associated with the method provides its mathematical underpinning, the notation provides syntactic sugar to make the theory more usable for developers. The B Method has much in common with the Z notation, mentioned above, and Abrial was also a contributor to that approach.

The B Method is described as a structured, rigorous development process. The basic structuring element of the B-Method is the B Machine which encapsulates state and behaviour in a modular, re-usable fashion. Rigorous proof can be performed to ensure that the specification is self-consistent, and consistent with other machines in the development. The B-method's Abstract Machine Notation (AMN) is used to describe the state, which is a mapping from variables to values; and behaviour, where operations are defined using the generalized substitution language. The notation provides features for deterministic and non-deterministic state transitions, for example assignment or choice. Non-determinism is useful at a high level of abstraction since it allows design decisions to be deferred, but the non-determinism is replaced by deterministic constructs

as development proceeds by way of refinement; and must removed completely before implementation. The invariant clause uses predicate logic to describe the properties of the system that must hold at all times, it can contain typing information for variables together with the desired properties relating to machine parameters, sets, constants and variables. The abstract machine specification is type checked to identify syntax errors, then proof obligations are generated. The proof obligations must be shown to hold for the system description to be self-consistent; it is preferable to discharge as many proof obligations as possible using an automatic prover, but in all but the simplest cases there are a number of proof obligations that remain to be discharged by hand. The user is able to guide the proof by suggesting strategies, and sub-goals in the form of hypotheses, in the endeavour to complete the proof. The B-method supports refinement, and an abstract machine can be refined several times leading to a hierarchical structure. Refinements are related to their more abstract counterparts in such a way that a valid refinement always satisfies a specification higher in the refinement hierarchy. The B method uses a refinement approach to add detail to a development, and proof techniques to establish that the a concrete refinement C refines an abstract specification A, we say $A \sqsubseteq C$. Informally we state that the concrete specification implies the abstract specification. Refinement generally takes two forms, with behavioural refinement operations are refined by weakening preconditions, strengthening guards and postconditions, and reducing non-determinism. In data refinement abstract data structures are given more concrete representations when moving closer to the implementation level of refinement. For example a development may use sets to represent data at a high level of abstraction, but a more concrete representation could be an array or a sequence depending on the implementation requirements. Tools supporting the B-method generate proof obligations relating to refinement, which must be discharged in a similar manner to those generated for proof of machine consistency. The tool with which we have the most experience is B4Free and the Click 'n' Prove interface described in [38].

The modelling activity makes use of mathematical concepts such as sets, constants and variables, which are underpinned by the set-theoretic representation. These features are used to describe the state of the system. Properties and invariants are based on first order predicate calculus and describe the constraints that must hold on the state at all times. Operations are substitutions that describe the changes of state, i.e. the variables of the machine, using AMN. The AMN is designed to have constructs similar to those used by programmers but it is, nevertheless, a modelling language. An example of a non-deterministic specification construct is the *ANY* statement,

ANY x
WHERE Q
THEN S
END

Here x is a parameter, Q is a predicate that includes the type of x, and S is a substitution. x is non-deterministically assigned a value satisfying Q. The statement is equivalent to the GSL definition $@x.Q \Rightarrow S$. The simplest substitutions are $SKIP$ which does nothing, and the assignment substitution, $x := E$; which states that x is assigned the value of the expression E.

Proof obligations are generated which must be discharged to show that the statements do not violate the invariant. The Classical-B approach is to find the weakest precondition required to satisfy some postcondition P. The proof obligation generated for the simple assignment is based on the substitution, E/x, where free occurrences of a variable x are substituted by the expression E. The rule for generating the weakest precondition follows,

$$[x := E]P = P[E/x]$$

and, for the ANY statement we have,

ANY x
WHERE Q
THEN S
END
$=$
$\forall x \cdot (Q \Rightarrow [S]P)$

2.1.6 The $^+$CAL Algorithm Language

The $^+$CAL algorithm language [96] is used to describe the behaviour of algorithms at an abstract level, and it is then translated to the TLA$^+$ [95] specification language for analysis. TLA$^+$ is a set-theoretic approach to system specification which is amenable to model-checking. The TLA$^+$ specification makes use of the TLC model checker to explore the state space and check all possible execution paths specified by the algorithm. Properties such as a non-terminating algorithm, or a deadlocking algorithm, are checked automatically. The user can also define properties that can be checked, such as the *invariant* or *assert* statements. The model checker ensures the invariant holds at all times, and ensures that an assertion holds when it is encountered on the execution path.

A $^+$CAL statement is a labelled atomic operation; and this is a concept that we make use of in the work described in this thesis. In $^+$CAL each statement must have a label, however the developer may omit it and allow the translator to add one automatically. The syntax for a statement is $LabelledS \ ::= \ [Label :] \ UnLabelledS$, where $UnLabelledS$ is an unlabelled statement; where brackets indicate choice of zero or one. The unlabelled statements contain a number of constructs, including assignment; looping and

conditional constructs such as $While$ and If statements; and also property specification statements such as $Assert$.

In $^+$CAL there are two equivalent specification notations, the p-style and c-style notations, with p-style being more verbose. We present the following p-style statement as an example specification showing the use of label annotations,

$$
\begin{array}{ll}
l1: & \textbf{if } a = 0 \textbf{ then } a := a + 1 \\
 & \qquad\qquad \textbf{else } l2: \ b := b + 1 \ ; \\
 & \textbf{end if} \ ; \\
l3: & x := 0 \ ;
\end{array}
$$

Notice that each label is associated with an atomic statement, and that the atomic statements are composed using the sequence operator ';'. Composition of atomic statements using the sequence operator is a feature of our work too.

2.2 Applying Formal Methods

2.2.1 State-based Methods and Process Algebras

We now consider the various approaches available to formal methods practitioners, in particular we can identify that individual formal methods lend themselves to solving particular types of problems. We can therefore categorize formal methods on the basis of the kinds of problems to which they may be appropriately applied (however the definition is not strict). In general we can identify state-based and process algebraic approaches. State-based approaches allow better descriptions of system state, whilst process algebraic approaches better describe behaviour of processes. State-based approaches tend to be more amenable to specifying properties related to allowable states and the transitions between them. Proving that a system satisfies the specified properties involves discharging proof obligations. Examples of state-based formal methods include B [5, 17, 44] and its successor, Event-B [6, 7, 8, 124], Z [143], and VDM [90]. Process algebraic approaches tend to be more suited to specifying behavioural properties of concurrent, distributed systems. A developer may wish to describe properties such as when transitions and communications can occur. Example of process algebras are CSP [78], CCS [115] and Pi-Calculus [116]. Generally it is easier to check liveness properties, such as livelock or deadlock, using a process algebra based model than a state-based model. A model checker, such as FDR [63], can be used to check these properties. Some approaches incorporate the notion of temporal logic, an extension of modal logic, to check whether properties hold with respect to time. For example it may be desirable to check if some property always, or eventually holds, or holds at a next

step and so on. Formal methods with these capabilities include TLA [95], Spin [79], SMV [109] or ASM [69].

2.2.2 Combining Technologies

It is often the case that process and state-based approaches are combined to take advantage of the benefits of both. An example is the combination of CSP and B found in [30], and similarly the $CSP \parallel B$, approach described in [131, 133]. Circus [159] is a combination of CSP and Z; and Alloy [85] bridges the gap between Z and object-oriented technology. In some cases graphical modelling approaches are combined with formal methods to aid the development process. The work on UML-B and the U2B translator [89, 110, 137, 138, 140, 142] established a basis for specifying B developments using a UML modelling tool; an updated version is part of the latest Event-B tool [141]. Similar work has been done to link XASM [15], an extension of ASM, to UML in [45]. One combined approach that is particularly relevant in our sphere of work is the combined approach using CSP and B [30, 35, 132]. We encountered the B Method in 2.1.5 so we proceed by providing some details of Communicating Sequential Processes (CSP). CSP was first introduced by Hoare in [78] and formalizes the behaviour of, and interaction between, processes. A CSP specification consists of a number of processes P and an alphabet of events related to the process, αP. The occurrence of events is described in a process specification, using the notion of prefix. A simple specification is $P = a \rightarrow Q$ where a is an event and P and Q are processes. This means that, assuming a is in the alphabet of P, the process can engage in the event a and then behave as Q. The notion of traces is introduced together with trace refinement. Properties of the system are described in terms of the traces, and a model is interpreted in terms of the failures and divergences of a system, based on traces. The failures of a system describe situations where no progress is made i.e. deadlock situations. The divergences of the system describe livelock; such as the situation where the system is exhibiting continuously looping behaviour, but is not communicating with the environment. It will therefore not make any $useful$ progress. The theory of CSP was revised in Roscoe's book [128].

In a development that uses a combined CSP and B approach the specifications are combined in such a way that synchronizes B operations and corresponding CSP events with the same name. The combined approach addresses recognised shortcomings in each of the approach. For instance CSP is a process algebra, its strength is in behavioural specification and it is weaker when used to describe state. The B-method's strength is describing the state of a system, and is weaker when it comes to the behavioural specification since it contains no high level constructs to compose events. In combining CSP and the B method the strengths of both approaches are harnessed. In the combined approach a CSP model is typically used to impose an ordering on occurrence of operations. We present a simple example to clarify this: we can describe a machine with variables

$v_1 \in \mathbb{Z}$ and $v_2 \in \mathbb{Z}$, and operations,

$$a \triangleq \textbf{BEGIN } v_1 :\in \mathbb{Z} \textbf{ END};$$
$$b \triangleq \textbf{BEGIN } v_2 :\in \mathbb{Z} \textbf{ END}$$

where $:\in$ is the non-deterministic assignment operator. The operations a and b are unguarded and we assume that the environment can non-deterministically select either of the operations for execution if they are enabled. To impose ordering on the executions using a combined CSP and B model, we introduce a CSP process P specified as follows,

$$P = a \rightarrow b \rightarrow P'$$

In the combined model a and b may only occur when it is enabled in both CSP and the B specification. In this case v_1 will receive it assignment before v_2 before moving on to the next process P'.

2.2.3 Tool Support

For formal methods to be used efficiently tool support is necessary to establish that the model's properties have been satisfied. The approaches can be categorized as theorem proving or model checking. In approaches that use theorem proving rules are applied to the specification which give rise to proof obligations that must be discharged. When all proof obligations have been discharged the model has been shown to be consistent. Model checkers establish correctness using state space searches. Model checkers will search the state space to find states where properties are not satisfied; an exhaustive search which does not find any property violations establishes that the model is consistent. In some cases formal approaches are amenable to both model checking and theorem proving, and may make use of tool support from one or more sources. Event-B can be model checked using ProB [104] within the RODIN toolset [153], and third-party provers can be added as plug-ins. ASM can be used with the PVS [122] and KIV [56] for theorem proving, or can be model checked [39]. The Alloy [84] tool supports the Alloy language, and makes use of the kodkod constraint solver [154]. CSP is used with the FDR [63] model checker, Spin is a model checker with its own specification language called Promela; and Z/Eves is a tool for developing and analysing Z specifications [129].

Each approach has its limitations, for instance model checking approaches can suffer from state space explosion where the number of states to be checked increases exponentially with the number of variables in the model. With general purpose proof tools such as PVS it is typically the case that they require a high level of expertise to use, and since they are not tailored to a specific formal approach they are not optimised for efficiency. Tools are often not integrated in a way that provides a seamless approach. The Rodin tool has been designed to provide much more specific support for Event-B than a tool

such as PVS; in this respect the Rodin tool's aim is to simplify the proof activity, and provide an integrated development tool to improve productivity.

As individual formal methods mature the proof tools that they make use of may change and suites of tools may be developed to support the development effort. The Event-B approach is integrated with the RODIN tool. This is designed to be an extensible platform, allowing for additional tool support, such as new proof tools, to be added by contributing a plug-in to the platform. VDM has a suite of tools known as VDMTools[59].

2.3 Object Oriented Technology

Object-oriented software engineering is a well established technique for organising the development and implementation of software systems. In its simplest form an object is simply an area in memory that contains some data and/or some code to perform some tasks. Usually, in a class based system, a class represents an abstract data type and is used to specify what data an object should contain, and what behaviour it should exhibit; instances of the class are objects that reside in memory. We use the terms instance and object interchangeably in this work, since they refer to the same thing, but generally we talk of instances when referring to objects of a particular class, and objects when we are being more general. The uptake of object-oriented technology in safety critical systems has been slower than in non-critical systems since the technology gives rise to some difficult issues such as how to handle (or whether to prohibit) dynamic binding and dynamic memory allocation; or how to manage inheritance, or complex control flows through multiple classes. In the field of avionics, however, a handbook arising from the *OOTiA* project [57], from the Federal Aviation Administration and National Aeronautics and Space Administration, highlights key issues and possible approaches for the use of Object-Oriented Technology in Aviation.

There are many object-oriented technologies including Java [67, 68], C++ [146], C# [119], Python [155] and Eiffel [111, 112] to name but a few. For the most part they all use similar concepts such as inheritance, polymorphism, and encapsulation. Inheritance and polymorphism are mechanisms that allow specifications to be re-used and tailored for specific uses. Encapsulation is the concept of limiting access to the data contained within an object in order to ensure that the state of the object remains consistent. In practice most object-oriented languages rely on the skill of the programmers and system architects to achieve a well encapsulated system. Ensuring that a system behaves as intended, in the light of concurrent executions for instance, is not a trivial task.

A simple Java class is specified as follows,

```
public class C {
 private int a = 1;
```

```
 public int inc(){ a=a+1; return a; }
}
```

In the class above we can see the features associated with encapsulation, the **private** modifier indicates that the field a is only visible from within the class definition itself, the **public** modifier indicates that the method *inc* and class C itself are visible from anywhere. We also see the type of the field a is restricted to integer primitive values and the method *inc* returns an integer primitive value. Inheritance would be used to make available the data and behaviour in class C to some subclass D using the following class declaration, **public class** D **extends** C .... In some cases the methods of a super class can be re-defined in the subclass, in our example if *inc* may be defined to increment the value by some other integer, this would be an example of polymorphism.

2.4 Formality and Object-Orientation

Object-oriented technology is commonly used in software development and its spread has been aided by productivity tools such as the UML [121]. The domain of business and safety critical systems has also been influenced by object-oriented technology, and in this section we discuss this issue.

2.4.1 Object-Oriented Formal Specification

Object-oriented technology can influence formal methods, where the methods themselves incorporate object-oriented features such as forms of inheritance. This is the case for VDM++ [46], Alloy [84], and Object-Z [135]. In the case of Event-B it contains no object-oriented features itself, it has however been tailored for use with object-oriented technology using UML-B; a combination of UML style diagrams and Event-B. UML-B provides a graphical modelling environment for Event-B, the ability to model classes and instantiation; and additionally provides an action and constraint language for UML-B called μB introduced in [139]. The current UML-B tool does not address issues such as concurrent execution of processes, or the gap between the formal specification and target implementation. In similar work a variant of the UML's Object Constraint Language (OCL) [120] is described by the Logic for Objects, Constraints and Associations (LOCA), in [99], and allows constraints to be imposed locally within an class, across a number of classes, or on associations between classes. The work focusses on the translation of UML class diagrams to B models, but also discusses translations to Java [67, 68] for implementation and SMV for model-checking. Another similarity between these authors' work and UML-B is the use of statechart diagrams, in [98], where they show how to use statechart diagrams to specify the state transitions of reactive systems, and a translation to B. Another approach to formalising object-oriented developments

uses the *Model Driven Approach* (in the sense of *Model Driven Architecture* [64]). The approach presented in [49] uses an object-oriented development with the aim of deriving an implementation in the C language [93] from a B implementation. This work does not however address the issue of concurrent processing.

Re-use of artefacts using inheritance mechanisms is one of the main advantages of object-oriented technology, but it does give rise to many challenges. Some of the issues arising from the use of these techniques including subclassing and polymorphism are discussed in [20, 113, 114]. Due to these complexities we consider re-use to be beyond the scope of our current work, and is not strictly necessary for the approach that we propose here.

2.5 Modelling with Event-B

In previous sections we have given an overview of the field from its foundations and discussed how formal methods and object-oriented technology have interacted. We now provide background in more detail on approaches that we draw from more specifically in our work. We begin with the Event-B method [6]. Event-B is a subset of the original B-method [5, 10, 38, 41] developed by J.R. Abrial. It is a set theoretic approach to software systems development. Event-B has a notation, methodology and tool support for rigorous development of software systems. The basic structural features of Event-B are contexts and machines. Contexts are used to describe the static features of a system using sets, constants, and the relationships between them. Machines are used to describe the variable features of a system in the form of state variables, and guarded events which update state. Builders within the development tools generate proof obligations which must be discharged in order to show that the development is consistent. The proof obligations generated in classical-B are often very complex, the Event-B approach results in simpler proof obligations as described in [72], since Event-B consists of a simplified action syntax which gives rise to simpler proof obligations. A further simplification was made by adopting an event-based approach, where each atomic event has a predicate guard and an action consisting only of assignment statements. Events correspond to operations in the B-method; operation specification was more complex and included constructs for specifying preconditions and return parameters; these constructs are not features of Event-B. Due to these simplifications (and more efficient proof tools) a large number of the proof obligations may be discharged automatically by the automatic provers. Where un-discharged proof obligations remain the user guides the interactive prover by suggesting strategies, and sub-goals in the form of hypotheses, in the endeavour to complete the proof.

Event-B supports refinement; a machine can be refined several times leading to a hierarchical structure. Refinements are related to their more abstract counterparts in such a way that a valid refinement always satisfies a specification higher in the refinement

hierarchy. Event-B tools generate proof obligations relating to refinement, which must be discharged in a similar manner to those generated for proof of machine consistency. In some cases we may model entities in an abstraction that are defined in the event parameters; and in the refinement these entities may be modelled using machine variables. It is desirable to link the parameters of an abstract event (since they disappear in the refinement) with their more concrete representation. To do this we provide a witness, using the $WITNESS$ construct a predicate is used to describe the relationship between an event parameter of the abstraction and a corresponding variable in the refinement. This is then used to assist with discharging proof obligations. It is often necessary to specify a linking invariant to describe the relationship between the variables of the abstract and refinement machines. Inspection of the proof obligations can assist in this task since some of the un-discharged proof obligations provide information about this link. Another feature of Event-B is the ability to refine one atomic event with a number of events, thus breaking the atomicity, as described in [32].

Event-B development begins with the abstraction of the observable events that 'may' occur in a system, which leads to a specification describing the state and behaviour of the system at a high level of abstraction. Event based modelling uses the notion of guarded events to describe the observable events. An event is said to be enabled when the guard is true, otherwise it is disabled. Typically when an enabled event fires some state update occurs, which is described by the event's action. The high level abstraction can be refined, possibly a number of times. At each refinement step new events and state information are added. The purpose of refinement in the Event-B method is to introduce more detail into the model and at the same time maintain the model's consistency. Eventually the model should describe the behaviour of the system at such a level of detail that an OCB model can be defined. The OCB model is subsequently transformed to an Event-B model that can be shown to refine the abstract development, and the target source code. An example of textual Event-B is shown in Figure 2.1. The context, named *exampleContext* has a set, A. Machine *exampleMachine* sees *exampleContext* to gain access to its contents. It has variables b and c which are typed in the invariant along with any additional constraints on the state. In this example b is typed as a powerset of A and c is an Integer. The example shows an event named *inc* which increments the value of c. The event named *new* non-deterministically selects an element of set $A \backslash b$ and adds it to b, using set union. Events consist of guards and actions, *inc* has no non-deterministic parameters, so the guard is in the *WHEN* clause, and the actions are in the *THEN* clause. Where the event has non-deterministic parameters, as in *new*, the guard is contained in the *WHERE* clause. Guards are predicates which describe the conditions under which the event is enabled, actions are substitutions which describe the effects of the event. An event describes the transition from the 'before' state to the 'after' state which happens atomically; that is, there is no intermediate state visible. In a consistent model the guards of an event ensure that its actions do not violate the invariant. When developing a software system it can be useful to view the occurrence

CONTEXT exampleContext
SETS A

MACHINE exampleMachine
SEES exampleContext
VARIABLES b, c
INVARIANT
b $\subseteq$ A $\wedge$ c $\in$ 0 .. 10
EVENTS
INITIALISATION =
b := $\varnothing$ $\parallel$ c :$\in$ 0 .. 10

inc =
WHEN c+1 $\in$ 0 .. 10
THEN c := c+1
END

new =
ANY x
WHERE x $\in$ A $\setminus$ b
THEN b := b $\cup$ {x}
END

FIGURE 2.1: Example of Textual Event-B

of state changes, in shared memory concurrent systems, as atomic events. We aim to relate the atomic state changes of such an implementation to atomic events described in an Event-B model, using OCB. This will simplify reasoning about the system under development since our abstraction does not include details of locking, unlocking and the implementation of conditional waiting. Event-B provides the formal semantics for the OCB notation which is introduced in subsequent chapters.

One important aspect of the Event-B approach is that *any* enabled event may occur, but only one of the enabled events may occur at any one moment. When modelling certain aspects of a system we may wish to impose an ordering of events. However, there is no sequence operator provided in the Event-B approach. It is therefore necessary to make appropriate use of guards and and state variables to model this aspect of a system. For example if we wish to impose an ordering on two events *evt1* and *evt2* so that *evt*1 occurs before *evt*2 we can use the following approach. Introduce an enumerated set $Grds = \{one,\ two,\ stop\}$ and a variable $step \in Grds$. Initially $step := one$; and we make use of *step* in the event guards as follows,

$$evt1 = \textbf{WHEN}\ step = one\ \textbf{THEN}\ \ldots \parallel step := two\ \textbf{END}$$
$$evt2 = \textbf{WHEN}\ step = two\ \textbf{THEN}\ \ldots \parallel step := stop\ \textbf{END}$$

This ensures that initially *evt*1 is enabled and *evt*2 is disabled since $step = one$; only

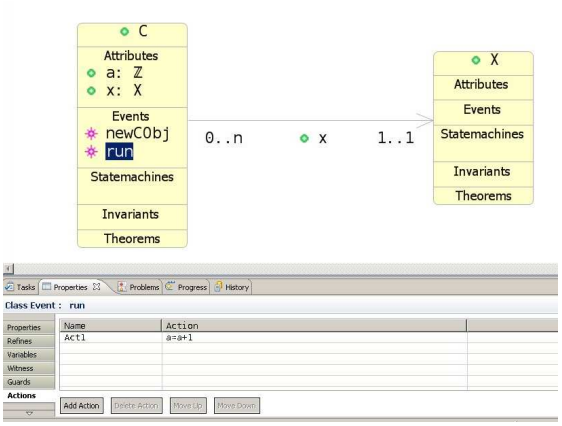


FIGURE 2.2: Class Diagram for UML-B

after *evt*1 has updated the *step* variable to *two* is *evt*2 enabled. At this time *evt*1 is no longer enabled since its guard is now false. Finally no events are enabled since $step = stop$ and all guards are false.

2.6 Modelling with UML-B

The formal modelling techniques of the previous section may be enhanced by the use of graphical modelling methods, [125, 126]. There has been much work on this topic with UML-B and the U2B translator, [110, 137, 138, 139, 140, 141, 142]. UML-B links the graphical modelling techniques and object-oriented features of the UML, to the B-Method and Event-B. We discuss here its application to Event-B in particular. UML-B uses UML-type visual modelling features such as package diagrams, class diagrams and statechart diagrams. However it should be noted that UML-B is not a UML profile for Event-B, it has its own meta-model. It does not use the UML specialisation features such as stereotypes, so in this sense it is only UML-like, but better suited to Event-B specification. One feature of systems that cannot be specified in UML-B, at the time of writing, is that of concurrency - in our work we wish to address the issue of concurrency (but not within UML-B itself). We find it useful to apply some UML-B modelling techniques with respect to modelling object-oriented features. An example UML-B class is shown in Figure 2.2 and its mapping to Event-B is shown in 2.3. In the Event-B model the formal representation of a class C is a carrier set C_SET representing all potential instances of C. The instances of C are modelled as a subset of C_SET. A constructor event adds a non-deterministically selected object, $newC$ from $C\_SET \setminus C$, to the set of instances C (The *class* X constructor is omitted). Class C has an attribute a of type $\mathbb{Z}$, which maps, in the generated Event-B model, to a variable a which is typed as a function $a \in C \rightarrow \mathbb{Z}$. Variable a relates the instances of class C to its attribute value, so an instance s of type C has an attribute value represented by $a(s)$. The variables representing instances are initially empty, no instances exist until the constructor is

CONTEXT ctx
SETS C_SET; X_SET

MACHINE mch
SEES ctx
VARIABLES C, X, a, x
INVARIANT
C ⊆ C_SET ∧ X ⊆ X_SET ∧
a ∈ C → ℤ ∧ x ∈ C → X
EVENTS
INITIALISATION =
C := ∅ ‖ X := ∅ ‖ a := ∅ ‖ x:= ∅

newCObj =
ANY self, x
WHERE self ∈ C_SET \ C ∧ x ∈ X
THEN C := C ∪ {self} ‖
x(self) := x ‖
a(self) := 0 ‖
END

run =
ANY self
WHERE self ∈ C
THEN a(self) := a(self)+1
END

FIGURE 2.3: Event-B Machine of a UML-B Model

called. The constructor models instantiation and initialisation of the objects. Events of a class are translated to Event-B events with the same name. We have shown a class diagram translated to Event-B; provision is also made for translating statechart diagrams to Event-B, transitions between states are translated to events which are guarded by control variables derived from the names given to the states. The association x enables the multiplicities of the instances to be specified. The owner of the association (origin of the arrow) has multiplicity $a \ldots b$, and at the opposite $c \ldots d$ (in Figure 2.2 $a = 0, b = n, c = d = 1$). Table 2.1 describes the relationships between association multiplicities, and the domain and range properties of the generated variables.

	a	b	c	d
0	non-surjective	-	partial	-
1	surjective	injective	total	functional
n	-	non-injective	-	non-functional

TABLE 2.1: Multiplicities of Associations

Package diagrams are the top level diagram used, these describe the relationships between models and contexts, and the refinement relationships between contexts, and

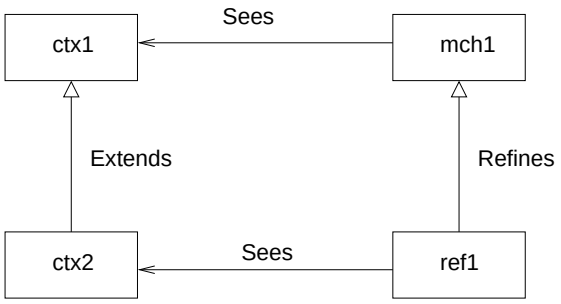


FIGURE 2.4: Package Diagram for UML-B

between models. An example package diagram can be seen in Figure 2.4 where machine $mch1$ sees $ctx1$, $ref1$ refines $mch1$, $ctx2$ extends $ctx1$, and $ref1$ sees $ctx2$. In order to describe event actions, and actions associated with transitions of a state machine, μB is used. μB is an action and constraint language for UML-B which is based on Event-B action syntax. It differs from Event-B syntax in a number of ways, for instance it uses the reserved word **self** which is a reference to the current object. It also allows an object-oriented style dot-notation for accessing instances referred to by attributes, so $i.v$ is equivalent to $self.i.v$ and refers to attribute v of the object associated with $self.i$.

2.7 The B0 implementation Language

The aim of formal specification of a system is usually to describe a system that will eventually be implemented. The step from the formal model to the implementation level is our area of interest. In previous work with Classical-B, B0 [43, 44] was developed which provides the final link between model and implementation. It is an implementation level language which can be translated to Ada [147], C and C++ source code for compilation into executable form; as far as we are aware no such approach exists for Event-B.

In a B0 implementation machine the OPERATIONS clause contains deterministic, concrete substitutions; and state is described using concrete data types that are amenable to implementation. The use of concrete substitutions at the implementation level means that all non-deterministic substitutions of the abstract development have been removed and can be implemented using a programming language. Therefore operations with preconditions and any other form of underspecification are prohibited. The use of concrete data refers to the fact that implementation level data structures must correspond to the structures that can be implemented using a programming language. B0 supports the following concrete data types: integers, booleans, arrays and records. The concrete substitutions are referred to as instructions, concrete predicates as conditions and concrete expressions as terms - in order to distinguish them from their abstract counterparts. Instructions include assignment, local variable declaration, operation call, branching and looping constructs, and each instruction is atomic. We present a small example of the

syntax to give a flavour of the style; beginning with the assignment (the becomes equal instruction)

$$
\begin{aligned}
&\mathit{Becomes\_equal\_to\_instruction} ::= \\
&\quad \mathit{id} := \mathit{Term} \\
&\quad \mid \mathit{id} := \mathit{Array\_expression} \\
&\ldots
\end{aligned}
$$

The variable named *id* is assigned the value of a Term or Array_expression. Terms are types such as an integer or boolean literal, arithmetic expression, or an identifier. Array_expressions can be one dimensional such as the following $arr_1 \in \{0..5\} \rightarrow \mathbb{Z}$, or of higher dimension such as the following two-dimensional array, $arr_2 \in \{0..5\} \times \mathrm{BOOL} \rightarrow \mathbb{Z}$. B0 allows atomic operation calls that return a value; the *operation_call_instruction* is defined as,

$$\mathit{Operation\_call\_instruction} ::= id_v \leftarrow id_{op}$$

where id_v is an identifier of a variable v and id_{op} is the name of an operation. Here the return value from operation id_{op} is assigned to the variable named id_v. In this case both the operation call and the assignment of the return value to the variable occur in one atomic step. B0 has a sequencing operator for instructions which permits sequential composition within an atomic clause. It is defined as,

$$\mathit{Sequence\_instruction} ::= \mathit{Instruction} \ ; \ \mathit{Instruction}$$

The *conditional_instruction* has optional **elseif** clauses and optional **else** clause. Conditions are a subset of classical-B predicates which have corresponding constructs in an implementation. Typical operators are $=$, $\neq$, $\leq$, $\wedge$ and $\neg$, which are used with simple terms such as identifiers, integer literals and boolean literals; or in the case of logical operators, conditions. The *conditional_instruction* is evaluated in a single atomic step. If the first condition is true then the first atomic clause of the instruction is evaluated, and the instruction completes. When a condition is false the next atomic sub-clause is evaluated and the instruction completes, and so on. Branches are evaluates until either the atomic **else** clause is evaluated, if one exists, or the instruction is completed with no updates. If no **else** clause exists this is equivalent to the sub-clause **ELSE** *skip*. The

instruction is defined as,

Conditional_instruction ::=
IF *Condition* **THEN** *Instruction*
[**ELSIF** *Condition* **THEN** *Instruction*]
[**ELSE** *Instruction*]
END

The *while* instruction is an atomic loop; that is, the whole loop completes in a single atomic step. The loop has a variant which must be shown to be decreasing; the variant is used to show that the loop eventually terminates. The invariant clause is used to type the variable used in the variant.

While_instruction ::=
WHILE *Condition* **DO** *Instruction*
INVARIANT *Predicate*
VARIANT *Expression*
END

It is intended that the OCB notation performs a similar role to B0, in that it links the formal model with a target programming language in order that an implementation be created. However, unlike B0, we support the specification of concurrently executing processes, and the sharing of objects. We will also be targeting the Event-B approach instead of classical-B. To support translation to an object-oriented languages, and to make use of encapsulated data, we introduce concepts to OCB such as class definitions; object instantiation; and atomic procedure calls.

2.8 The Java Language Specification - Second Edition

This section deals with the Java Language Specification - second edition (JLS 2) [37] which serves to define the language up to version 1.4 of the Java SDK. Our initial investigations begin with this as the target platform, and we begin the section by describing Java with this version in mind. The third edition (JLS 3) [68] is introduced towards the end of this thesis in Chapter 8 when we investigate how we can leverage the more recent additions to the Java language.

2.8.1 Programming with Java

The Java language is the object-oriented programming language of the Java Platform, it is typically used to specify the behaviour of applications that will run on a Java Virtual

Machine (JVM). It is strongly typed, and incorporates mechanisms to allow concurrent processing, using the Java notion of threads. A Java program is usually translated to machine independent bytecode, and a platform specific JVM interprets the bytecode. This machine independent approach is described as 'write once run anywhere'. A Java program may however be compiled to a native executable form - this is platform specific, so it loses 'write once run anywhere' capability. Portability and cross platform support is seen as one of the main benefits of Java over other programming languages like C [93] and C++ [146]. Java also contains simplifications that help to reduce some of the errors that can be introduced to C and C++ programs, such as the notion that developers do not have access to pointers; but this is achieved at the expense of some flexibility.

Other benefits of using the Java platform are that it has multi-threading features built-in, these features are extremely useful - a number of threads can be created, and these threads are able to progress or wait depending on whether certain conditions are satisfied. This can make for more efficient use of system resources, but does introduce a number of problems which will be discussed further in this chapter. Java uses the notion of synchronized monitors to achieve mutually exclusive access to shared data. When a synchronized region is defined a lock will be obtained before the region is entered. The use of these features is not enforced in Java however, so mutual exclusion is not guaranteed simply by the use of synchronized regions alone, extra effort must be expended to ensure ***all*** accesses to a particular piece of data obtain the lock. Use of these locks can cause threads to deadlock in certain situations and we discuss this further, later in the chapter.

We have already mentioned that Java is a class-based, object-oriented language, the simplest class definition, class $ClName$, resides in a file of the same name, $ClName.java$, and the file resides in a *package* which is used to restrict the scope of visibility. A typical class definition is,

public class ClName{< *body* >}

The body of the class can contain, field declarations, method declarations and other class declarations. The **public** modifier indicates that the class is visible from any package. If the modifier was not present then the class would only be visible from within a package. Similar scoping rules apply to fields and methods. **private** is a modifier to restrict visibility to within the declaring class; and **protected** is used to restrict visibility to the declaring class and any subclasses. Inheritance is one of the main re-use mechanisms of Java and classes are said to *extend* superclasses, indeed all classes defined in Java extend the universal superclass *java.lang.Object* by default. We declare an extension as follows where $ClName$ inherits features of $SuClass$,

public class ClName **extends** SuClass{< *classBody* >}

Using this declaration *SuClass*' **public**, and **protected**, fields and methods are inherited by *ClName*. Additional fields and methods can be declared in *SuClass*, and overriding can be used, where new or additional behaviour can specified for an existing method. Within a redeclaration of an inherited method (including the constructor method) **super**() can be used to invoke the superclass method of the same name.

Methods in Java can contain local variable declarations. The scope of these variables is the lifetime of the method invocation and they are stored in stack memory rather than in heap memory as fields are. Declaration of method parameters also gives rise to local variables. A typical method declaration is as follows,

public synchronized void anOp(< *paramList* >){< *methodBody* >}

As before we use the **public** modifier to indicate that the method has public scope, visible anywhere. The **synchronized** modifier is used to indicate that a monitor lock must be acquired before method entry, locking is discussed in more detail in following sections. The **void** modifier is used to declare that the method should not return a value. A return type, either a primitive such as an **int** or a reference type, can be specified here.

Threads are Java's mechanism for the lightweight scheduling of executions. A class that is required to run as a thread inherits from either *java.lang.Thread* or *java.lang.Runnable*, Threads can be declared using the extension mechanism discussed above, or by an alternative mechanism - interfaces. Threads can be declared by specifying that they should implement the *Runnable* interface. Interfaces are a flexible mechanism used for multiple inheritance. They are used to declare class variables, and method signatures that its subclasses must implement. In our work we will use the *Runnable* interface, which we use in the following way when wishing to declare a class *P* exhibiting thread behaviour,

public class P **implements** Runnable{< *classBody* >}

Class *P* implements the *Runnable* interface; this entails providing an implementation for a *run* method in < *classBody* >.

public void run(){< *methodBody* >}

A thread can then be created and started using the following fragment based on the above.

```
Runnable p = new P();
new Thread(p).start();
```

The **new** keyword is used to create new instances, in this case we create a new instance of P assigned to p and pass it to the *Thread* constructor. The *Thread* class' *start* method is invoked which in turn calls P's *run* method.

2.8.2 Concurrency, Interference and Locking

We now discuss some of the pitfalls that developers should be aware of when programming using JDK 1.4 (defined in JLS 2). The problems arise in part because of the complexity that concurrency introduces to a development, and partly due to ambiguities and shortcomings in the Java Language Specification itself. Lea notes, in [100], that the concept of concurrency in difficult to pin down. In summary: Apparent and real concurrency in Java [68] is provided by the Java Virtual Machine (JVM) and the underlying Operating System. Java uses the notion of threads; a number of threads may be in the running state at any one moment in time. The Java Memory Model allows for a range of possible configurations, from one thread per processor on a multiprocessor, to many threads per processor, and of course a single thread. The illusion of concurrency is achieved by time-slicing when the number of threads exceeds the number of processors. The JVM has a scheduler that allocates an amount of processor time to each thread and decides when each should start and stop processing, each running thread uses processor resources independently of other threads but may share main memory with other threads. The scheduler selects threads for execution according to some arbitrary strategy, and no fairness guarantees are provided about selection of individual threads for execution.

The Java memory model defines the relationship between reads and writes of memory values and how to achieve consistency in the face of data races and optimisation. One such problem occurs with the synchronisation of the CPU registers and cache (the working memory), with main memory. According to the memory model each thread can potentially run in its own CPU. Certain compiler optimisations can be performed whereby data values may not be written to main memory but remain in the CPU cache. These optimisations can lead to a thread having an inconsistent view of the state, and therefore unintended behaviour may occur when applications access shared data in an unrestricted manner; the main (shared) memory may not contain the latest updates. In addition to these issues which can cause problems for the developer there are a number of ambiguities related to the specification of the memory model itself. Some of these problems are highlighted by the authors of [71] where they describe the Java memory model of the Java Language Specification [67] using ASM. They highlight the problematic nature of interpreting the specification and the well known problems with suspending and resuming threads. An additional problem with JLS2 is that of accessing *volatile* fields. By declaring a field *volatile*, accesses to the field by different threads are mutually exclusive, but do not require a synchronized block, or method. The intention was to ensure

that read accesses always provide the most up-to-date value. The JLS specified that reads and writes of *volatile* fields should be made directly to main memory and that caching should be prohibited. Additionally it specified that each thread's actions on a *volatile* field should be performed in the order that they were requested, and said nothing about *non-volatile* fields. This was sufficient for ensuring consistency of single threaded programs, but did not consider the potential problems with multi-threaded programs sharing data. The problem arises since *volatile* and *non-volatile* reads and writes can be re-ordered by the compiler. Therefore, although accesses between *volatile* fields remain ordered, accesses between *volatile* and *non-volatile* fields may be interleaved. Therefore between different threads the re-ordering is visible and can cause unintended behaviour.

To a large extent the Java Language specification - third edition [68] (JLS 3), which coincides with the release of Java SDK 5, addresses many of these issues by redefining the memory model. We use JLS2 in our initial approach since the technology was mature, and its problems were well understood, at the time we began the work described in this thesis. In our work initial work we make use of the simplistic synchronization mechanism, which corresponds well with Event-B's event-based approach. However, the use of the synchronization mechanism turns out to quite restrictive, and we will also explore the use of JLS3 to overcome these restrictions.

When developing sequential (single-threaded) Java programs where the result conforms to the Java memory model, there is no possibility of data being observed in a state that is not consistent with the intended semantics. This is because the state is not observable externally, and any compiler optimisation that takes place will preserve the internal consistency, i.e. consistent intra-thread semantics. Compiler optimizations may include re-ordering of execution at bytecode level, or omission of writes to main memory.

The introduction of multiple threads to a development does not automatically introduce interference problems, but there are certain conditions under which it may occur. Interference can be defined as data being observed in a state that is not consistent with the intended semantics, the cause of which is an update by a concurrently running thread. If threads do not share state, either directly or indirectly, then no interference is possible. In that case threads can be thought of as independent, sequential programs obeying as-if-serial semantics. Whenever state is shared between threads in an uncontrolled manner the potential for interference is introduced and the program execution may depart from its intended (as-if-serial) semantics.

Uncontrolled interleaving of Java threads may cause problems due to the relationship between Java source code and bytecode. A simple assignment such as, $x{=}x{+}1$, in Java may be compiled to a number of lines of byte code. The bytecode may read x, add the value and write the new x value in a number of discrete steps. The bytecode is able to interleave with another thread between the read, add or store instructions. Values may then be read or updated which may not be consistent with as-if-serial semantics.

It is even the case, for long and double types, that higher and lower order bits may be written at different times, so a read access may obtain a completely meaningless value with updated low order bits and the previous value of the higher order bits. These problems are compounded by the fact that the updated values of x may never be written to main memory but may be retained in CPU cache due to optimisation.

To address the problem of visibility of inconsistent state Java provides a built-in synchronization mechanism which has effects at both high and low levels of implementation. The synchronized keyword is used in the method header to indicate to the compiler that main memory must be synchronized with the CPU's working memory. Working memory refers to the caches and registers of a CPU. Synchronization takes place upon acquisition and release of the synchronization lock, at the beginning and end of a synchronized method. Upon lock acquisition values are loaded from main memory to the CPU's working memory, upon lock release cached values are flushed from working memory to the main (shared) memory. The built-in Java synchronization mechanism also provides mutual exclusion at a higher conceptual level, since only one calling thread can be associated with a lock at any one time. Atomic behaviour can be achieved through appropriate use of locking and encapsulation. However, the use of synchronized methods and blocks is not enforced. This means that the development can contain errors which are difficult to detect, an issue which is addressed by our approach.

The synchronized keyword is used to identify a method or block of code that needs to access an object without interference from another thread. Java uses the synchronized keyword to facilitate the use of monitors [77]. Each instantiated object has a monitor with a lock. When a thread attempts to call a target object's synchronized method it must first obtain the monitor lock for the target object. If the lock is not available the Java Virtual Machine (JVM) blocks the thread. It places the calling thread in the target object's lock wait set, with any other threads that have previously made unsuccessful attempts to obtain the lock. These blocked threads will be made runnable (removed from the lock wait set) by the JVM when a thread exits a synchronized block or method. Threads in the runnable state are then available for execution, and may be selected at some arbitrary time, by the JVM, for execution. In the case where the lock is successfully obtained at the first attempt, execution of the method continues without blocking. When returning from the synchronized method the lock is released and the JVM makes threads in the lock wait set (if any have joined) runnable.

2.8.3 Conditional Waiting

In concurrently executing systems a particular thread may have to wait for some condition to become true before being able to proceed. A typical example is of a producer and consumer where producer threads and consumer threads share a buffer; if the buffer is full the producer should wait until the buffer has space, if the buffer is empty then

the consumer must wait for data to arrive. If the threads were to continuously loop and check for the arrival of data the processor would be continuously engaged in this activity. A more efficient method is to cause the thread to wait until the associated state changes. When the state changes, the thread is notified and can check to see if it is able to proceed. While the thread is in the waiting state it is not consuming processor resources. The term conditional waiting is used to describe this behaviour (introduced in [77]), where a thread waits until some condition becomes true, at which time execution may proceed into a conditional critical region (CCR). Classes used in Java inherit (directly or indirectly) from the *Object* class, this class makes available the basic features required for conditional waiting. The operations *wait*, *notify* and *notifyAll* are part of the interface inherited from the *Object* class. The following Java fragment shows the structure required to implement conditional waiting, (however exception handling is omitted for clarity). The operation is guarded by an entry condition *enterCCR*. The operation causes the calling thread to wait if the entry condition, *entryCond*, is not true. If the entry condition is true, execution may proceed into the CCR.

```
public synchronized void anOp(){
    while(!enterCCR){
            wait();
    }
    ... /* Conditional Critical Region */
}
```

Consider a thread t and shared object o. A thread t may invoke *o.anOp()*. Thread t must successfully acquire o's monitor lock (prior to method entry). The *wait* call is contained within the body of a while loop which continues to loop, and wait, until the entry condition *enterCCR* is satisfied. The thread will remain in its waiting state until notification is received from another thread using *notify* or *notifyAll*; or it may wake when a specified period of time has elapsed. When the entry condition is satisfied the execution will exit the while loop and enter its CCR. The *wait* call causes the calling thread, t, to be added to the conditional wait set of the object, and o's monitor lock is released. Thread t is now in the waiting state where it could remain indefinitely. To avoid this a parameter may be supplied with the *wait* call, the parameter specifies a time after which t is made runnable again. Thread t can also be removed from the conditional wait set by another thread that invokes o's *notifyAll* method. Thread t is now runnable and may be selected for execution by the JVM. When execution resumes it does so by returning from the *wait* method, so the loop condition will be tested again. To simplify our approach we will ignore the timed wait; and we leave the discussion of exception handling until we deal with the implementation specifics. For now we just note the fact that the *wait* method may throw an *InterruptedException*, which must be handled or propagated. This exception will occur if a waiting thread is interrupted, by a

call to its *interrupt* method. Notification is used when some state is updated for which other threads may be waiting. A Java fragment showing the call follows,

```
public synchronized void update(){
    ...some update
    notifyAll();
}
```

Consider a thread t and shared object o. When thread t invokes *o.update()*, *notifyall* is called which causes all threads in o's conditional wait set to become runnable. If there are a number of threads waiting for notification then any one of them may win the race for execution. An alternative approach is to use the *notify* method but this can introduce problems and its use is not recommended in most situations. A *notify* call will make runnable only a single, arbitrary, thread in an object's conditional wait set. In some situations using *notify* will fail to wake the thread that will allow execution to proceed. In general the *wait*, *notify* and *notifyAll* methods must be placed inside a synchronized method, failure to do so gives rise to an *IllegalMonitorStateException* at run-time; the calling thread must own the monitor lock before calling *wait*. The synchronization mechanism is used to provide mutually exclusive access to data and is also used in conditional waiting. However, due to the fact that its use is not enforced, inconsistent state may be visible. In the work presented in this thesis, we bring formal methods to bear on the problem; we develop a new approach which hides the locking and waiting implementation issues from a developer by relating atomic constructs in an intermediate specification (model) to a formal, Event-B model. The atomic constructs of the intermediate model are reflected in the Java implementation, but the insertion of the Java constructs is handled by automatic translation from the intermediate model to Java, thereby relieving the developer of this task.

2.8.4 Looping and Branching Problems

Consider the following pseudocode fragment (which is slightly unrealistic but used to demonstrate a point). Suppose a class accesses a public variable a. The public visibility declaration means that a can be accessed from outside the class in which it is declared. In our example a class, C is a sub-class of *java.lang.Thread* which runs as a thread, and may be one of a number of threads that have access to a. In the following fragment a is read, and then division is performed (which is undefined if the value is zero).

```
if(a>0){ result = n/a; }
```

In the following scenario we can see how interference occurs, where variable a is accessed by thread *t1* of class C, and thread *t2* which is of some other class decrements the value

of a by one. In the following scenario variables a and a' are the before and after values respectively, and the initial value of a is 1.

step	t1	t2	a	a′	effect
1	a>0		1	1	condition is true
2		t1.a=t1.a-1	1	0	decrement a
3	result =n/a		0	0	error

The error occurs because the attribute a is decremented by *t2* after the condition evaluation and causes a divide by zero error. More generally values that are inconsistent with the intended semantics may be read when there is a dependency between the condition and the body. The condition and body are not evaluated atomically and the read and write are conflicting actions. In order to prevent this kind of problem it is necessary to restrict access to fields by making them private, thus restricting visibility to within the class only. Access is then permitted only through the use of synchronized methods, which enforces mutually exclusive access.

2.8.5 Deadlock

Another issue that arises in the discussion about concurrency in Java is deadlock. The Java environment provides the conditions for deadlock, i.e. a locking mechanism which allows a thread to hold locks while waiting for others, and no pre-emption of resources competing for the same lock. A circular wait occurs in the following scenario which leads to a deadlock. *t1* and *t2* are threads that need to acquire both locks a and b.

1. *t1* obtains lock a
2. *t2* obtains lock b
3. *t1* attempts to obtain lock b,
   - but blocks waiting for b (also holding lock a)
4. *t2* attempts to obtain lock a,
   - but blocks waiting for a (also holding lock b)

The threads *t1* and *t2* are blocked, forever waiting for the other's resource to be freed. A solution to the problem is to provide a feature which allows a thread to give-up locks that it holds in the event that it discovers one of the locks that it wishes to acquire is not available. Since it can be very difficult to know in advance which combinations of acquisitions will lead to deadlock a cautious approach would lead us to use a non-blocking locking strategy whenever we need to obtain more than one resource within a synchronized method or block. This approach may however lead to livelock; a situation where the threads are continuously looping and finding that the resources are unavailable, even though they are not unavailable all of the time.

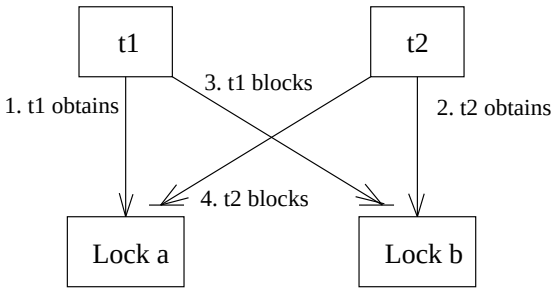


FIGURE 2.5: Occurrence of Deadlock

An alternative strategy, to overcome contention that gives rise to deadlock, is to impose an ordering on lock acquisitions. An example of this is given in [25] where a type system is specified that incorporates the specification of lock levels and specification of a partial order between each lock level. Where a thread holds more than one lock they must be acquired in descending order. The type checker checks that the implementation does indeed preserve the ordering. A similar strategy, acquisition of locks in a pre-defined order, could be used in our approach. Lea describes alternative locking strategies in [101] which forms the basis of the java.util.concurrent.locks API. It is an implementation of one of these strategies that we later use in Chapter 8 to implement the lock manager for our transactional constructs.

2.8.6 Nested Monitor Problem

Another problem that affects the strategy employed in our approach is the nested monitor problem, which can occur when a thread holds more than one monitor lock and a wait is invoked. In the following scenario of Figure 2.6 a thread acquires two monitor locks and is then caused to wait. Unfortunately the thread still holds one of the locks so if this object is required to make the waiting threads runnable, deadlock will occur. Here we have two threads, *t1* and *t2*, that make calls through synchronized *Admin* methods. These methods, in-turn, call synchronized methods of *anObject*. One method contains a conditional wait and another contains *notifyAll*. The following scenario leads to *t1* holding onto *admin*'s lock, when it is has been put into the waiting state.

1. *t1* obtains the admin lock.
2. *t1* obtains anObject's lock - is caused to enter the waiting state, releases anObject's lock but retains admin lock.
3. *t2* is the only thread that can wake *t1* but it is blocked.

t1 is waiting in *anObject*'s conditional wait set. If another thread were to invoke a method that issues a *notifyAll* call then *t1* would be made runnable and possibly relinquish all locks held. The problem arises if *t2* is the only thread that can cause the notification to be issued, and *t2* has to go through the *admin* object which is held by *t1*.

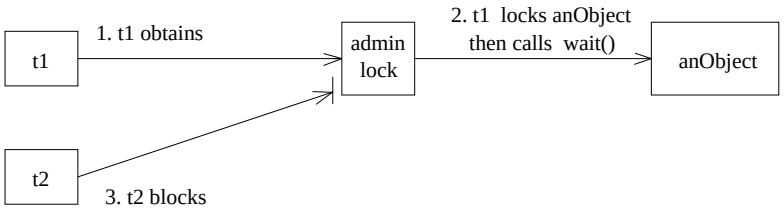


FIGURE 2.6: Nested Monitor Deadlock

In this case *t2* is blocked indefinitely waiting for the admin lock, *t1* is waiting indefinitely for notification that *t2* would provide if it were not blocked.

2.8.7 Formal and Semi-Formal Approaches for Java Implementations

We have seen some of the problems arising from the use of Java implementations, in particular when concurrent processing constructs are used. Much work has been done to address these issues and we now provide an overview of some approaches seeking to address these issues that are of interest to us. An approach addressing concurrency issues is that of JCSP [157, 158], which establishes a link between CSP and Java. A fuller account of JCSP is given in Section 2.9.1. JCSProB, described in [162], makes use of the JCSP libraries. The ProB tool can be used to construct and model check a combined CSP specification and B machine, which can then be translated to Java code. The work extends the JCSP libraries to accommodate external choice and message passing that includes data structures. The FSP notation and LTSA tool used in [108] focusses on the modelling of Concurrent Java systems, and subsequent verification of the model using the analyser. Several toy examples are presented but the tool falls short of automatic translation to Java. The work of Jacobs et al. [87, 88] uses annotated Java source code to make dependability claims about concurrent object-oriented programs, it describes concepts such as packing and unpacking objects, and object-invariants. The resulting formalisation is verified using a bespoke tool. Other work that attempts to clarify the issues surrounding concurrency using a clear view of atomicity is Fondue [94]. It is a notation that uses OCL (but is not specifically linked to Java) for the specification of concurrent programs. Eiffel [111, 112] is an object-oriented programming language that additionally contains formal specification constructs known as assertions, such as pre and post-conditions, class invariants, loop variants and loop invariants. It is recommended that updates to system state are performed by commands; and state interrogated by queries. The separation of concerns, where commands do not return a value and functions do not update state, is known as command-query separation. This approach is however not enforced in Eiffel.

Some formal methods incorporate the Java Modelling Language (JML) [28, 102] or OCL [120] into their specifications. The JML is a markup language used to annotate

Java source code. The markup uses constructs such as invariants, preconditions, postconditions, atomic regions and locks to specify required behaviour that enables verification by model checking (but fails to verify all of the atomicity requirements as found in [127]). JML also allows insertion of runtime assertion checks. The Key system [24] uses UML diagrams annotated with OCL, or JML, to produce Java code. The underlying formalism is based on a Dynamic Logic called Java Card DL, which underpins translation to Java Card implementations. PerfectDeveloper [55], described in [47], is an approach that uses its own notation, in a design-by-contract style, for specifying object-oriented developments without concurrent processing. The specification gives rise to proof obligations which are discharged by an automatic theorem prover; there is no option to perform interactive proof; the tool targets Java or C++ [146] code. Other work is aimed at extending the existing Java language, such as the atomic type system presented in [62], which proposes the addition of *atomic*, *guarded_by* and *requires* keywords, to the Java language. The type system is expressed formally, and maps to a Java application that type checks the annotated code (checking for atomicity violations in the specification).

2.9 Java Correctness and Concurrency

The following section describes some of the approaches which can be used to improve the dependability of Java implementations. Java will be the target platform for our implementations, a decision influenced by its widespread use, and ability to accommodate concurrent processing. Of the many methods available we have chosen to give details of these in particular since they give a valuable insight to the alternative approaches to achieving improved reliability.

2.9.1 JCSP

JCSP [157, 158] is a Java library arising from a combined CSP/Occam model, providing a framework for implementing the *Occam* [134] approach to concurrency. It uses a message passing, rendezvous style, as a basis for communication between concurrent Java threads. [123] extends the theory to distributed networks of threads. The approach is underpinned by CSP semantics and makes use of the notions of processes and channels to provide a point-to-point communication style. A CSP specification of JCSP has been verified using the FDR model checker. A monitor abstraction is used to describe the behaviour of synchronized Java methods. Lock acquisition and release are described in the following CSP processes $STARTSYNC$, and $ENDSYNC$. The *claim* and *release* events represent lock acquisition and release respectively, and o is the object being locked

by thread t.

$$\begin{aligned} STARTSYNC(o.t) = \ & claim.o.t \rightarrow SKIP \\ ENDSYNC(o.t) = \ & release.o.t \rightarrow SKIP \end{aligned}$$

The processes that model locking are used with the $MLOCK$ and $MLOCKED$ processes which provide an ordering of events for modelling waiting, as follows,

$$\begin{aligned} MLOCK(o) = \ & claim.o?.t \rightarrow MLOCKED(o,t) \\ MLOCKED(o,t) = \ & release.o.t \rightarrow MLOCK(o) \\ & \Box\ notify.o.t \rightarrow MLOCKED(o,t) \\ & \Box\ notifyAll.o.t \rightarrow MLOCKED(o,t) \\ & \Box\ waita.o.t \rightarrow MLOCKED(o,t) \end{aligned}$$

We can see that once the lock is claimed the $MLOCKED$ process can choose to release the lock, perform notification, or wait. The CSP semantics of JCSP are described in [157]. Processes are implemented using the CSProcess class which contains a *run* method to initiate communication between threads. The process communicates with other processes only through *channels*, or *events*. The JCSP *One2OneChannel* is used to facilitate communication between two processes, other configurations for differing numbers of readers and writers are provided by, *Any2AnyChannel*, *Any2OneChannel* and so on. The channel classes typically have a field to store the transmitted value, a boolean field indicating if the channel is empty, and two methods *read* and *write*. The correspondence between the JCSP classes and the CSP model allows the behaviour to be checked for issues such as deadlock using a tool such as FDR [63]. Various *Occam* features are found in JCSP such as external choice, and parallel and sequential composition of processes.

2.9.2 JCSProB

A recent development has been the combination of JCSP and ProB [104] called JCSProB [162]. The ProB model checker used in JCSProB supports a combined classical-B/CSP model [35] albeit with some restrictions applied. Its translation utility creates Java code based on the JCSP framework. The approach is restricted to a single CSP-machine pair and does not feature refinement. In a combined B and CSP development events of the CSP processes synchronize in the normal way, and when combined with a B machine, synchronize with CSP events with the same name and parameter types. The effect of this to facilitate an ordering on the operations of the B machine. A combined operation/event with the same name and parameters is only enabled when CSP events are enabled and the guards of the operation are true. The operation signature $o = o_1, \ldots, o_m \leftarrow op(i_1, \ldots, i_n)$ is related to the CSP statement $ch!i_1 \ldots !i_n?o_1 \ldots ?o_m$.

We see that the CSP model sends its parameters to the B machine and receives the result from the B machine return parameters. The following CSP process PROC restricts the B machine of 2.1.5 to performing op_1 followed by op_2 and then op_1 again.

$$PROC = op_1!x_1?y_1 \rightarrow op_2!x_2?y_2 \rightarrow PROC$$

JCSProB provides a specialized channel class *PCChannel* for the Java implementation. The class inherits from java.lang.Thread, so implementations override the *run* method to describe state updates. Four other operations implement the behaviour which includes the possibility of parameter passing between the CSP model and B machine. The $void\ ready()$ method is used where there are no parameters to pass, $void\ ready(Vector\ in)$ is used where the CSP process passes parameters to a B operation, $Vector\ ready\_rtn()$ is used where a CSP process receives arguments from a B operation, and $Vector\ ready\_rtn(Vector\ in)$ where a CSP process passes parameters to B operation, and receives return parameters back from a B operation. A precondition check can be performed in the $preConditionCheck()$ method, it blocks the calling thread if the precondition is not satisfied. The combined model is amenable to model checking with ProB and can be used to check safety and deadlock properties.

2.9.3 JML

The Java Modelling Language (JML) [28, 102] is a semi-formal method based on the design-by-contract (DBC) [111] approach. In DBC a client of a service is expected to satisfy the preconditions of a contract in order to use a service safely. In return, the service provider's activity is described by the postconditions of the contract; the provider is expected to satisfy the postconditions. In a markup language such as JML the contract is at a higher level of abstraction than an implementation, this allows the developer flexibility to define more than one implementation satisfying a single contract. In this way implementations can be changed without affecting the specification, for instance to provide a more efficient implementation. JML allows a developer to annotate Java programs with markup contained in comments /*@ *markup* @*/, or following //@ so that traditional compilers ignore the additional information, expressions are similar in style to Java expressions in order to make them understandable to those familiar with Java; however JML assertions are not allowed to have side effects. As in Java, a JML specification be inherited from a superclass, and similar scoping rules can be applied to the JML specification itself. Properties specified using JML can, most often, be checked mechanically. The keywords *requires* and *ensures* describe the precondition and postcondition respectively and are the foundation of its DBC style. An example of

their use follows,

```
public class Div{
  int val;
  //@ invariant 0 <= val;
  /*@ requires divisor! = 0;
    @ assignable val;
    @ ensures val == \old(val)/divisor
    @ && \result == val;
    @ signals(DivideException) val/divisor < 0
    @*/
  int divide(int divisor){
    if(val/divisor >= 0)
      val = val/divisor;
    else
      throw new DivideException("Value to low");
    return val;
  }
}
```

In the above example we see the JML specification in comments which include an invariant. The invariant states that *val* must always be greater than or equal to zero, and if the invariant is violated during the method call an exception should be signalled. It is possible to specify two kinds of postcondition, normal and exceptional. The *signals* keyword allows identification of exceptional postconditions, and the *ensures* clause allows specification of normal postconditions. In the case of method specifications, they are typically followed by the Java code that implements the method specification. The *requires* clause defines the precondition, the client will have to check that $divisor! = 0$. The implementer of the service must guarantee that *val* is changed and assigned the new value, as described in the *ensures* and *assignable* clauses. The $\backslash result$ clause states which value should be returned by the method. In some situations it is necessary to distinguish between the $pre-state$, the state before entering a method; and $post-state$, the state on method exit. This can be seen above where $\backslash old(val)$ is used to access the old value of *val*. Another JML feature is the ability to use quantification when specifying properties, $\backslash forall$ is used for universal quantification, and $\backslash exists$ is used for existential quantification.

There are a number of tools available that can be used with JML [29]. The JML specification can be compiled using the *jmlc* compiler, which converts pre-conditions into Java assertions and adds them to the Java bytecode; these can then be used to perform

run-time assertion checks provided that assertion checking is enabled in the JVM. Static checking and verification of assertions is possible using other tools. ESC/Java2 is a static checker that has been extended to check JML annotations against Java code. The tools create verification conditions from the annotations and code, as well as verification conditions and makes use of a automatic theorem prover to find errors. The checker is typically used to discover out-of-bounds array references, non-null references - and for concurrent programs, deadlock and race conditions. ESC/Java2, however, is not used to 'prove' absence of all errors, but simply that none have been found by the checker. A more interactive approach, is to use the LOOP tool [26] to create a specification suitable for use with the PVS theorem prover [122].

2.9.4 Java Pathfinder

Java PathFinder (JPF) [103, 156] is an approach where Java bytecode is model checked directly using a modified Java Virtual Machine (JVM); it is particularly useful for checking for concurrency issues such as deadlock and data races. The authors argue that model checking at the bytecode level provides a number of benefits, including the fact that instructions in the bytecode are easy to handle in the analysis tool. It is also possible that the application of formal methods at the design level does not prevent some errors being introduced at the lower, implementation, level. Therefore model checking at this lower level can eliminate these errors, however it is still the case that correct bytecode does not imply that the Java is correct. As is common with model checking approaches JPF suffers from state-space explosion as the complexity of the model increases; several techniques are described to reduce this problem such as symmetry reduction and static analysis. Symmetry reduction seeks to prevent areas with similar behaviour being re-checked, by deriving a canonical form and checking this just once; JPF uses this technique in relation to class loading for instance. Static analysis uses slicing and partial order evaluation techniques, among others, to reduce the state space. Slicing will typically reduce the state space by selection of a subset of the state space that is relevant to a particular property under review. Partial order reduction (POR) techniques try to identify interleaving/non-conflicting areas in the code which can be reduced to a single execution path; thereby reducing the number of paths to analyse. In the current tool POR can be carried out on-the-fly.

JPF checks the bytecode to find unhandled exceptions, data races and deadlocks in the code. To find (but not prove absence of) potential data races it uses an algorithm based on the Eraser algorithm of [130].

The main advantage of JPF is that it works on Java bytecode without the need for the developer to learn a new language, but there is currently no support for many native code libraries such as java.awt, and only limited support for java.io, Java's I/O libraries. The

Model Java Interface (MJI) is a partial solution to this problem, it allows a developer to replace a code library with an abstraction.

2.9.5 JR - extended Java

JR[92] is an approach to parallel distributed computing which can also be used to specify executions in shared memory systems. This work is interesting since it describes an approach to handling concurrency which uses an extension to the Java language, which is then translated back to a standard Java program to provide an implementation. JR introduces a version of Java which is extended with the SR [13] concurrency model (Synchronizing Resources). Initially we say a few words about SR before moving on to JR. The SR concurrency model allows specification of one or more Virtual Machines (VM) in an environment with one or more physical machines; each VM resides on, at most, one physical machine and a physical machine can contain more than one VM. Processes can share objects within a single VM or across a number of VMs. Each VM has *global* and *resource* objects, these have a two distinct partitions - a public and a private segments. The publicly visible part of the specification contains only type, constant and operation declarations; additionally *globals* may declare variables. The 'private' part is the body, which contains private declarations and the operations' behavioural specification. The variables declared in a body are visible only from within the body in which they are declared. In SR the units of execution are *processes* and *procs*. *processes* are created and run when the enclosing resource is created; whereas *procs* are invoked explicitly by a caller. In SR processes communicate using operation invocation and this can take one of several forms, we discuss this further when dealing with some of the Java programming aspects.

JR introduces a number of communication styles at a high level of abstraction. In traditional Java programming Remote Method Invocation (RMI) is the highest level of abstraction used for communication in its distributed model. Indeed the JR implementation uses RMI in its implementation, but it is hidden from the developer. The caller of the operation can use a synchronous or asynchronous communication style, and the operation can be serviced using a procedural style, or alternatively using an invocation queue. Synchronous style communication is invoked using an operation *call*, asynchronous communication uses a *send* request. Operations are serviced as follows,

- Procedure Invocation Style - an operation is serviced by an object inheriting from *ProcOp*, the operation is simply invoked by executing the method body.
- Invocation Queue Style - this is derived from the SR *input* statement (a generalisation of Ada's *select* and *accept* [147]), and manifests itself as an invocation queue in JR. Operation invocation requests are queued in the called object. Operations

wait in the invocation queue to be serviced by the *inni* statement, in this style the server chooses when to invoke the operation.

When the different communication and servicing approaches are combined they give rise to the following communication styles of Table 2.2. The first column of the table shows the communication style from the caller, which is either a synchronous call (caller waits for reply) or an asynchronous send request (caller does not wait for a reply). The second columns describes the way in which a receiver can service the communication, which is either in a procedural style (no queue) using *ProcOp*, or by queuing requests in an invocation queue where the server imposes its scheduling a policy. The third column is descriptive, but it is worth noting that an asynchronous procedure call is described as dynamic process creation; by this we mean that a new process is created and run in order to service the request.

Communication	Service	Style
Synchronous	ProcOp	Procedure Call
Synchronous	InOp	Rendezvous
Asynchronous	ProcOp	Dynamic Process Creation
Asynchronous	InOp	Message Passing

Table 2.2: Communication Styles of JR

In summary - JR introduces some useful extensions to Java which, at a high level of abstraction, allow specification of distributed, concurrent systems. Features include the creation of remote virtual machines, without the need for user configuration at run-time; remote object/process creation; support for asynchronous and synchronous communication styles; two methods of servicing - invocation queue and direct invocation. JR extended-Java is translated back to a standard Java format for implementation.

2.10 Review of the Chapter

In this chapter we have provided an overview of the domain in which we make our contributions. We began by introducing some formal theories that underpin the state of the art in our field of interest; we continued with a general discussion about the various development approaches that can be used to improve the reliability of systems. We discussed how formal methods tend to be applied to particular kinds of problems, and described several approaches that combine formal methods to provide broader solutions. Next we introduce object-oriented technology and discuss how object-oriented technology has influenced specification in the formal methods field. The next section introduces the Event-B formal method which forms the basis for the work described in this thesis. The UML-style graphical interface for Event-B specification, UML-B, was then introduced. It is from UML-B that we take inspiration for some of our modelling techniques.

This was followed by a description of the B0 implementation language, which is the implementation notation associated with classical-B development. We then discussed some problems encountered in Java implementations related to the use of concurrent executions, and sharing of memory. We concluded with a discussion of some approaches aimed at improving the specification, and reliability of Java implementations.

Chapter 3

The OCB Language Part 1 - Processes and Monitors

In this chapter we introduce OCB with a discussion of our motivation and the main aspects that influence our approach. We introduce OCB in an incremental way, beginning with non-atomic constructs, followed by monitors and procedure calls. Much of this chapter, and indeed the chapter that follows it, is related to and expands upon our paper [54].

3.1 Motivation

When modelling a software system, an Event-B model will be refined to a point where we are ready to provide information about the implementation. Our motivation then is to provide an approach to link Event-B with an implementation that takes advantage of multi-threading by allowing executions to interleave at certain points. Consideration is given to how tasks may be performed by executing processes (granularity of atomicity); and how to specify where the processes may interleave, and the pitfalls of interleaved executions in an environment using shared memory; how to create an approach to specifying systems that contain features of the Event-B formal method and also of object-oriented technology.

We shall use Java [68] as the target implementation language since it is often used to implement concurrent systems; however our work is not limited to this target in principle. We introduce an intermediate specification language, Object-oriented, Concurrent-B (OCB), which we use to link Event-B models and object-oriented implementations (see Figure 1.1). The new notation sits at the interface between the two technologies, and we incorporate aspects of both. From OCB we define two separate translations; the first, OCB to Event-B, gives rise to an Event-B model and thus embodies the semantics of

the OCB model. The second, OCB to Java, gives rise to Java code that implements the OCB model. We would expect to show that the Event-B model refines an existing model in order to show that it satisfies properties of some abstract model. We aim to have a notation which abstracts away some of the implementation detail from the developer, and provides a simple view of atomicity with which to reason about the system under development. We use labels to identify atomic steps, similar to those in $^{+}$CAL [97], which we map to program counters.

When defining the mapping to Java we need to ensure freedom from interference by restricting visibility of data, and enforce a mutual exclusion policy for access to shared data. We also utilize conditional waiting, but incorporate restrictions to avoid the nested monitor problem [105] (where a monitor incorrectly retains a lock when a thread waits). In particular we are concerned with preventing interference between concurrently executing processes. Concurrent execution of interleaving processes is a typical way of scheduling activities in a system where, using time slicing, each process can periodically undertake some of its processing. Interference can occur when processes share memory and values observed by a process are changed unexpectedly by some other process. A process running in isolation from other processes is said to have as-if-serial semantics. When a process is subjected to interference it deviates from its as-if-serial semantics as described in [100], and we need to prevent interference in our implementation.

3.2 An Introduction to OCB

We begin by discussing some of the key issues, and the overall strategy for our approach. An Event-B model may consist of a number of events which are abstractions, that when implemented, are able to run in an environment that supports concurrency. Each event of the abstract development can contain a number of updates to the state, which occur atomically. Using OCB we impose an ordering on the interleaving atomic steps, and translate this to an Event-B model which refines the abstract development. The events of the implementation refinement are restricted so that they occur in the order specified in the OCB specification. To facilitate the interleaving behaviour we introduce a sequential operator,';', and the notion of non-atomic operations, running in parallel, which may interleave at the point of the sequential operator. To accommodate concurrency within our system we introduce processes. A process' behaviour is described by a non-atomic operation. A non-atomic operation consists of one or more labelled atomic clauses where the labels map to program counter values in the Event-B model. The program counters are used to guard the events, and impose an ordering on the execution of the clauses of each process. In our system we wish to share data between the processes in a controlled way, to do this we introduce monitors with atomic procedures. Access to monitor variables is restricted in such a way that processes can only access the shared variables through atomic procedure calls. We also add the restriction that monitors are

not able to call a process', or another monitor's, procedures; thus preventing the nested monitor problem.

3.2.1 Java and Event-B

Naturally, when attempting to create an interface between two technologies such as Java and Event-B we find that some desirable, and perhaps some undesirable characteristics manifest themselves. We are attempting to create a specification notation at a higher level of abstraction than Java and find that we can make use of Event-B's notion of atomicity. At the same time we find that we are constrained somewhat, in order to avoid some of the problems discussed in Chapter 2.

Atomicity

Developers creating concurrent programs using a language such as Java bear the responsibility of ensuring their code is thread-safe; that is, the concurrently executing threads obey as-if-serial semantics. It is widely recognised that reasoning about these aspects of concurrency is very difficult, mistakes are easily made and errors are hard to detect. We simplify a developer's task by allowing them to reason about concurrency at a higher level of abstraction. Event-B facilitates this and leads us to a solution where the Java developers use a simple notion of interleaving atomic clauses. The developer, using a higher level approach, is therefore relieved of the burden of specifying the lower level locking details. An intermediate specification notation arises, with high level atomic constructs, which maps to atomic events in Event-B. The notation is used to describe a formal model of an object-oriented program. The new OCB notation includes object-oriented features to facilitate mapping to object-oriented languages. We further propose a mapping to Java code, where the resulting program has the same atomicity as the corresponding Event-B model.

Event-B has a very simple notion of atomic events. Unlike unprotected Java methods, intermediate states in an event are not visible. There are discrete 'before' and 'after' states associated with a state transition. The state transition occurs when an event fires and the actions of the event determine the new state. This simple concept of an event is represented by an atomic clause in the OCB model; the Java program, that results from the transformation, is made to reflect this granularity of atomicity. For shared data, in the translation, we map an OCB procedure call (and its related event) to a synchronized method. Unshared data does not require protecting in this way. In the Java implementation the Java bytecode generator adds additional code - on entry to a synchronized method; prior to method entry, the monitor lock is obtained, the critical activity is performed, and the lock is relinquished upon method exit. This sequence relates to the formal model when an event takes place. The synchronized method will

only be atomic if certain restrictions are observed, such as respecting encapsulation. These restrictions must be enforced by development processes and tools to ensure code conforms to the formal model.

Concurrency

In Event-B two events cannot take place simultaneously so there are no concurrency operators involving events, although Event-B can be used to describe concurrently executing systems. It also does not contain object-oriented features. It is, however, possible to construct an Event-B model that represents an object-oriented system that is partitioned (into threads/processes) and keeps track of states of multiple threads. In such a model the finest granularity of the interleaving is determined by the Event-B semantics for events, where an event is atomic. In Java unrestricted interleaving can cause unpredictable results and is difficult to reason about. So the relationship between events and suitably protected Java code will provide a clear definition of atomicity for the Java implementation. It is our intention that the atomicity of the Java program will correspond to the atomicity of the formal model, i.e. the interleaving of the Java threads can be shown to be equivalent to the interleaving of the formal model with its atomic events. The relationship between OCB specifications, the Event-B model, and the resulting Java program will need to take into consideration interference and deadlock. Our current work imposes restrictions on an attribute's visibility to prevent interference, and each atomic clause is restricted to interact with a single monitor object. This will ensure freedom from unintended deadlock caused by contention for multiple resources.

3.2.2 Process and Monitor Classes

In order to simplify reasoning about the interactions between interleaving threads we stipulate that only non-shared objects can map to threads in the implementation. In OCB this gives rise to the non-shared thread-like objects that we call processes, which are instances of $ProcessClass$; and shared objects which are instances of $MonitorClass$, which we refer to as monitors, see Figure 3.1. The relationship between the processes and shared objects is similar to the relationship between Ada tasks and protected objects. Ada tasks access a protected object's data using its procedures, functions and entry calls; these features ensure tasks have mutually exclusive access to shared data. $ProcessClass$ instances correspond to 'runnable' threads and each $ProcessClass$ has a *run* operation

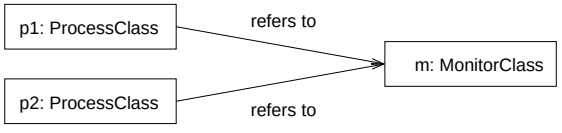


FIGURE 3.1: Processes Sharing a Monitor Object

which is non-atomic. The *run* operation consists of a non-atomic construct which is able to interleave in a controlled manner with other *run* operations. The constructs have precisely defined points at which threads may interleave. In the Java translation of a *ProcessClass* the resulting process object is an instance of *java.lang.Runnable* (the interface for Java threads), and has a *run* operation containing a translated non-atomic construct. The non-atomic construct is defined recursively, so it may contain other non-atomic constructs. Types of construct include branching ($if-then-andthen-end$), and looping ($while-do-andthen-end$). It would be possible for *ProcessClass* definitions to contain atomic procedure definitions providing they are non-blocking, however we refrain from adding this feature at this moment in time in order to simplify the explanation - we intend to add this feature as part of future work. Instances of *MonitorClass* are the servants of processes and do not map to threads, they translate to monitor objects in Java. Each *MonitorClass* contains only private data and atomic procedure definitions. These procedure definitions may make use of the conditional waiting construct; however recursive calls are prohibited in OCB due to the fact that correspondence with Event-B would be difficult to achieve.

3.2.3 Restrictions Required for Mapping to Java

To ensure access to variables is free from interference in the implementation we impose a number of restrictions. The operations of a *ProcessClass* may invoke *MonitorClass* procedures; and a number of process objects may share a monitor object. As mentioned earlier in the chapter we stipulate that access to shared variables be through atomic procedure calls, and add the restriction that monitors are not able to call a process', or another monitor's, procedures; thus avoiding the nested monitor problem.

In the translation to Java, our monitor classes should map to classes where mutually exclusive access to data is enforced by encapsulation, together with the use of synchronized method calls. The constructs that give rise to mutually exclusive access are added automatically by the tools during the translation process. To simplify our strategy we ensure that all of our methods are *synchronized*. While this is not very efficient, further optimisations can be investigated as part of future work; for example operations accessing attributes that are not updated after instantiation would not need to be synchronized. In order to ensure that the classes arising from the translation are properly encapsulated we require that all OCB attributes give rise to private fields. They are therefore not directly accessible from outside of the declaring class, and we enforce the use of synchronized methods to access data.

We now discuss the issue of integer wrap-around in Java. An integer's value, in Java, ranges from -2147483648 to 2147483647, and arithmetic operations on Java integers are modular. In many cases wrap around will not cause a problem since the integer values will be nowhere near the minimum or maximum value; or perhaps wrap-around

behaviour in some unusual circumstances is part of the specification. If we wish to prevent it then it is possible to define a constant named *JavaInteger* in the context; the range is defined in an axiom with the interval $-2147483648 \ldots 2147483647$. Then we can use *JavaInteger* when typing the integers instead of $\mathbb{Z}$ which will give rise to additional proof obligations. Our default for this work is to use $\mathbb{Z}$ to define integers, and note that wrap-around is ignored for now, but the issue will be addressed in future work.

3.3 OCB Language Features

3.3.1 The Sequence Operator

In a system where concurrently executing processes are able to interleave it is necessary to describe the points at which the interleaving may take place, since uncontrolled interleaving may lead to inconsistent state being visible. In order to facilitate interleaving we introduce non-atomic operations, in process classes, which allow interleaving to occur at pre-determined locations. The locations at which processes may interleave are identified by a semi-colon character, which is a sequence operator indicating left to right ordering of evaluation. This kind of feature could be realised by either a pre-emptive or co-operative multi-tasking implementation; but since we are targeting Java we use its pre-emptive multi-tasking approach. The sequence operator's operands map to labelled atomic events.

3.3.2 Labelled Atomic Constructs

The atomic sub-clauses of non-atomic operations are identified with a unique label - when translated to Event-B each label corresponds to a state in which an event is allowed to fire. The model has an abstract program counter which keeps track of the current state. We also use the label to create a unique event name, this allows us to quickly relate events to the labelled clauses when reading the Event-B model. The event is enabled when the program counter is in the state identified by the label, providing of course all other guards are true. A simple example OCB fragment follows, where the labelled constructs have an assignment *Action*.

$$label1 : a := a + 1;$$
$$label2 : b := b + 1$$

3.3.3 A Looping Construct for Processes

Since looping is a frequently required behaviour, we wish to introduce a construct that is easy to reason about; we also impose restrictions on the loop condition to ensure that no other process can interfere with its value. We refer to the OCB looping construct as the **while** construct. In the **while** construct the guard refers only to private attributes of the current class. The processes are not shared so the attributes of this class will not be modified by another process. The **andthen** clause of a loop can contain a further non-atomic construct. The OCB *while* clause is written as follows, with the square brackets indicating an optional clause,

while(*Guard*) **do** *Action* [**andthen** *NonAtomic*] **endwhile**

The loop is guarded by a predicate *Guard*. While the guard evaluates true then the *do* clause is evaluated, and the *andthen* clause is evaluated if one exists. The looping behaviour continues until the guard is false.

It is important to note that this construct allows interleaving (with other threads) to take place within the loop body. The *Guard* and *Action* of the *do* clause are evaluated atomically, and this is optionally followed by a non-atomic clause in which interleaving may take place. We consider that breaking the atomicity of the loop will allow for easier proof, and makes for a more flexible approach. It is often the case that, during the execution of a while loop, a number of procedure calls may be required. In OCB we are limited to only one procedure/create call per atomic clause due to the restrictions we have imposed to prevent deadlock. Using the *andthen* clause we are able to overcome this limitation and allow two or more calls in a loop body.

3.3.4 Conditional Branching for Processes

Process class operations may specify conditional branching using the **if** construct. We have a similar restriction on attributes used in the guard as for the while loop, that is, attributes are private and non-shared. The **if** construct follows,

if(*Guard*) **then** *Action* [**andthen** *NonAtomic*] **endif**

There are additional branch options with an **elseif** and **else** clauses, they be discussed more fully later in the chapter.

3.3.5 Conditional Waiting for Monitors

In our initial work we use Java's built-in facilities for conditional waiting, *java.lang.Object*'s *wait* and $notifyAll$ methods. We allow monitor objects to block until a guard is satisfied. The OCB **when** clause, located in a monitor's procedure definition, will translate to these Java constructs. The conditional waiting construct has the following form, where < and > indicate syntactic elements,

$$\textbf{when}(< Guard >)\{\ < Action >\ \}$$

The atomic Conditional Critical Region (CCR), the *Action*, is executed when the guard is true. When the guard is false the operation blocks. In the Java translation the blocking behaviour corresponds to either unsuccessful monitor lock acquisition, or successful lock acquisition followed by failure to satisfy the entry condition. An unsatisfied entry condition causes the calling thread to block and the monitor lock to be released. In our approach waiting threads are woken by $notifyAll$ calls, which causes them to resume and compete once more for the monitor lock. This corresponding Java conditional waiting construct is as follows and may require an optional $notifyAll$ method call if other waiting process need to be informed of a state update that has occurred in the action.

while(!<Guard>){wait();} <Action>; notifyAll();

3.3.6 The MainClass Construct

The $MainClass$ is the analogue of a Java class with a *main* method. This provides the point of entry for execution in the model and facilitates the mapping to the Java class which contains the *main* method. The *main* operation of our $MainClass$ contains construction clauses for process and monitor objects. In the construction clause new instances are created. Process objects may refer to monitor objects; the name of a monitor object can be passed to the constructor operation as a parameter, the monitor object name is then assigned to an instance attribute in the constructor for later use. When a $ProcessClass$ is instantiated it is immediately available for scheduling, but may not run for some time. In the Java implementation, the scheduling of a thread occurs some time after the constructor call, the translator inserts a call to the Thread object's *start* method immediately after the thread is constructed, when it actually runs is decided by the scheduler.

3.4 Mapping Processes to Event-B

We begin our formal description of OCB by describing the non-atomic, and labelled atomic, constructs of processes. These are used to specify the behaviour of a process,

we also give details of how the labels are used to describe the states used in program counters. A system may have a number of processes definitions, each with a non-atomic operation that is able to interleave with non-atomic operations of other processes. We use a syntax based on the guarded command language [52], discussed in section 2.1.2; and later we use syntactic sugar to provide a more object-oriented style specification. Here we use ';' as the sequence operator, [] for choice, and *do od* for repeating. We also use the following BNF style annotation, where s is a symbol; $[s]$ denotes zero or one s is permissible; s^+ denotes 1 or more s is permissible; and $s*$ denotes 0 or more is permissible.

$$
\begin{aligned}
&\mathit{NonAtomic} ::= \\
&\quad \mathit{NonAtomic}\ ;\ \mathit{NonAtomic} \\
&\quad |\ \mathit{NonAtomic}\ []\ \mathit{NonAtomic} \\
&\quad |\ \mathit{do}\ \mathit{Atomic}\ [;\ \mathit{NonAtomic}]\ \mathit{od} \\
&\quad |\ \mathit{Atomic}
\end{aligned}
$$

The syntax of a non-atomic clause allows a sequence, choice, loop or atomic statement. Atomic statements have a body, optionally guarded by a predicate. The body may be an *Action* involving assignment, assignments are of the form $x := E$, where x is a variable name and E is an expression. They may be composed using, $\|$, the parallel operator. In our definitions of the OCB syntax we use left and right brackets, '$\triangleleft$' and '$\triangleright$', to delimit atomic regions. The syntax of the *Atomic* construct follows,

$$\mathit{Atomic} ::= \quad \mathit{Label} :\ \triangleleft\ [\mathit{Guard}\rightarrow]\ \mathit{Body}\ \triangleright$$

We present a simple example to illustrate the mapping of a sequential clause which gives rise to two Event-B events, $evt1$ and $evt2$. **WHEN** G **THEN** S **END** is the guarded event syntax of Event-B with guard G containing a predicate, and body S containing assignment actions. The labels of the specification map to values assigned to the process' program counter variable, P_{pc}. An example specification is,

$$l1 :\ \triangleleft\ y := x\ \triangleright\ ;\ l2 :\ \triangleleft\ x := x + 1\ \triangleright$$

which results in the following two events,

$$evt1\ \triangleq\ \textbf{WHEN}\ P_{\mathrm{pc}} = l1\ \textbf{THEN}\ y := x \parallel P_{\mathrm{pc}} := l2\ \textbf{END}$$

$$evt2\ \triangleq\ \textbf{WHEN}\ P_{\mathrm{pc}} = l2\ \textbf{THEN}\ x := x + 1 \parallel P_{\mathrm{pc}} := \mathit{terminated}\ \textbf{END}$$

The event $evt1$ is enabled when the program counter is $l1$. The state updates are contained in the event body, together with the program counter update where the value is set to $l2$; the next label in the sequence. The next event $evt2$ is enabled when the

program counter value is $l2$, which has been set by the assignment in the action of $evt1$. Once again state updates specified in OCB are translated to the event body. During translation we supply the value of the next label, in this case we specify that the process terminates and assign the label *terminated*. Each process in the system will have such a terminating label, although there may be some situations where translation does not give rise to its use, such as when specifying a process with a loop that is forever true. It should be noted that the above Event-B fragments use the $\triangleq$ operator in event definition. In this thesis we will however simply use $=$ in place of $\triangleq$ in the event specification, where $\triangleq$ has already been used.

The processes of a system are defined as a function over class names ($CNames$) to processes.

Definition 3.1. $Processes = (CName \rightarrow Process)$

A process is defined as a set of variables and a non-atomic clause,

Definition 3.2. $Process = \mathbb{P}(Var) \times NonAtomic$

The variables of the resulting Event-B model will include all variables of the translated process together with a separate program counter variable for each process.

We introduce a transformation function, TP, which maps a process' non-atomic clause to a set of Event-B events. TP is typed as follows,

Definition 3.3. $TP \in Processes \times CName \rightarrow \mathbb{P}(Event)$

In order to define TP we introduce a function TNA that maps a non-atomic clause to a set of events. The label supplied to TNA is the last program counter value assigned in the clause. TNA is typed as follows,

Definition 3.4. $TNA \in NonAtomic \times Label \times CName \rightarrow \mathbb{P}(Events)$

We denote function application, where we apply function f to one element d, as $< d >^f$ or to a number of elements $< d_1, \ldots, d_n >$ as $< d_1, \ldots, d_n >^f$. We now define the application of TP to a process with variables var, body na and name P. TP maps to a set of events, where tp is a constant label indicating a termination state for a process.

Definition 3.5. $< var,\ na,\ P >^{TP} \triangleq < na,\ tp,\ P >^{TNA}$

We see that the translation of the set of variable declarations var, associated with the process, is not addressed here. We wish to focus on the translation of the non-atomic clauses at this point in time. We simply state that each variable declaration map directly to an Event-B variable and typing invariant. An example declaration $a \in \mathbb{Z}$ gives rise to a variable a and an invariant $a \in \mathbb{Z}$.

Non-atomic clauses can be nested within other non-atomic clauses forming a hierarchical structure, the top-most level has no following clauses, so the label supplied to TNA at the top level is the terminating state *tp*. In order to define the non-atomic syntactic elements we specify some well-definedness constraints regarding labelling of the constructs. Non-atomic clauses are well-defined if the start label of each operand differs; the exception to this is the choice construct, where each label must be the same. To identify the first label of a clause we introduce a function *sLabel* (for start label), which takes a non-atomic construct as its input parameter and yields the first label of a non-atomic clause.

Definition 3.6. $sLabel \in NonAtomic \rightarrow Label$,

Applied to the non-atomic and labelled atomic clauses we have,

$$\begin{aligned} sLabel(l : \ \triangleleft\ [g\rightarrow]\ b\ \triangleright\ ) &= l \\ sLabel(na1; na2) &= sLabel(na1) \\ sLabel(na1 [] na2) &= sLabel(na1) = sLabel(na2) \\ sLabel(do\ na\ od) &= sLabel(na) \end{aligned}$$

We now look at sequential composition of non-atomic clauses; $na1$ and $na2$ are sequentially composed clauses. The label, $l2$, passed to the TNA function is the end label of the sequence. It specifies the final program counter value for the non-atomic clause.

Definition 3.7.
$$\begin{aligned} &< na1\ ;\ na2,\ l2,\ P >^{TNA} \\ &\triangleq\ < na1,\ l1,\ P >^{TNA} \\ &\quad \cup < na2,\ l2,\ P >^{TNA} \end{aligned}$$

where $l1 = sLabel(na2)$

A branching clause is defined as follows with $na1$ and $na2$ being composed using the choice construct. Note the mapping of different label parameters on the right hand side of the equality, when comparing to the sequence clause definition 3.7.

Definition 3.8.
$$\begin{aligned} &< na1\ []\ na2,\ l2,\ P >^{TNA} \\ &\triangleq\ < na1,\ l2,\ P >^{TNA} \\ &\quad \cup < na2,\ l2,\ P >^{TNA} \end{aligned}$$

In a well-defined branching clause the guards of each branch, and any sub-tree, are disjoint. Labels play an important role in determining the execution order in the translated Event-B model. The branching clause maps to two TNA transformations, where $sLabel(na1) = sLabel(na2)$; and both *Label* parameters are the same. This contrasts with the transformation of a sequence clause where $sLabel(na1) \neq sLabel(na2)$. In a sequence clause, the *Label* parameter of the first clause equals the start label of the second clause, that is $l1 = sLabel(na2)$, in order to model the enabling conditions for ordered

execution. However, in a branching clause the start labels form one of the enabling conditions used to define choice between branches.

Now we turn our attention to the looping clause, the body of the loop consists of a *Body* clause, and optionally a non-atomic clause. The definition of the simpler case, without the optional non-atomic clause, follows,

Definition 3.9. $< do\ l1:\ \triangleleft\ g \rightarrow b\ \triangleright\ od,\ l2,\ P >^{TNA}$
$\triangleq\ \{< l1:\ \triangleleft\ g \rightarrow b\ \triangleright\ ,\ l1,\ P >^{TLA}\}$
$\cup \{< l1:\ \triangleleft\ \neg g \rightarrow Skip\ \triangleright\ ,\ l2,\ P >^{TLA}\}$

Clause $l1$ is guarded by g; if g is true then b occurs, the program counter is unchanged and the loop body can be evaluated again. In the case where the guard is false the action is *Skip*, and the program counter is set to the value supplied as the *Label* parameter. We now present the mapping where the optional non-atomic clause, na, is present. In the following definition the program counter is updated to allow evaluation of na using the label identified by $sLabel(na)$. The last event arising from the clauses of na resets the program counter to the initial value, this models the behaviour where the loop can begin again, or exit depending on the guard.

Definition 3.10. $< do\ l1:\ \triangleleft\ g \rightarrow b\ \triangleright\ ; na\ od,\ l2,\ P >^{TNA}$
$\triangleq\ \{< l1:\ \triangleleft\ g \rightarrow b\ \triangleright\ ,\ l3,\ P >^{TLA}\}$
$\cup \{< l1:\ \triangleleft\ \neg g \rightarrow Skip\ \triangleright\ ,\ l2,\ P >^{TNA}\}$
$\cup < na,\ l1,\ P >^{TNA}$

where $l3 = sLabel(na)$

Transformation of a labelled guarded atomic action is defined next. The transformation TLA takes an atomic statement, the end label and owning process name as parameters, and returns one or more events. If the guard is omitted from the specification then a true guard is assumed.

Definition 3.11. $TLA \in Atomic \times Label \times CName \rightarrow \mathbb{P}(Event)$

The label of the clause forms part of the event guard, and the end label supplied to TLA is the updated value of the program counter used in the action. The name of the event is derived from the label and caller, and if necessary the clause type, here $l1_{\mathrm{P}}$ is P's label $l1$. We define the transformation of an atomic clause with a body consisting of action A, as follows.

Definition 3.12. $< l1:\ \triangleleft\ g \rightarrow A\ \triangleright\ , l2,\ P >^{TLA}$
$\triangleq$
$l1_{\mathrm{P}}\ =$
WHEN $P_{\mathrm{pc}} = l1 \wedge g$

THEN $A \parallel P_{\text{pc}} := l2$
END

where P_{pc} is the program counter of the process P.

Note that assignment actions in A may be composed in parallel. In this case we apply the restriction that a variable used in A may only appear on the LHS of an assignment once.

3.5 Mapping Monitors and Procedure Calls to Event-B

We now introduce monitors and procedure calls to the system. Monitors are shared resources which enforce mutually exclusive access to their variables through atomic procedures. Our system now has non-atomic process bodies, atomic procedure calls, and atomic assignments. Procedures can have formal parameters, which we define as a sequence, $LVar$, of parameter declarations. Within a monitor class the names of formal parameters must be distinct from the attribute names, this is due to the substitutions that take place during translation, a restriction that may be overcome if required in the future. The sequence of formal parameters correspond with the sequence of actual parameters in a call. Translation of a procedure call results in the in-line substitution of the procedure body in the caller, in place of the call; and formal parameters are substituted by actual parameters. Substitution of formal parameters by actual parameters is described in [117, 118]; we use substitution by value but limit use of formal parameters to the right hand side (RHS) of assignment expressions, and to guards. The procedure name is unique in a monitor, but the same name may exist in another monitor. We therefore need a way to link the called procedure with the appropriate monitor; we use dot notation to do this. This is similar to the dot notation that we use later for object-oriented features; except that here, on the left hand side (LHS), we are identifying a monitor name; and later the name of an instance appears on the LHS. To be well defined the monitor must contain a procedure with the called name and with the same number of parameters. For each of a call's actual parameters, $a_1, \ldots, a_k$, the type must match those of the formal parameters, $f_1, \ldots, f_k$. To enable the specification of a return parameter we introduce a special variable with the reserved name, **return**, that can be used in an action clause. The **return** variable can only be used on the LHS of an assignment statement in the procedure body. When in-lining it will be substituted by the variable assigned to on the LHS of the procedure call. The syntax for the body of a labelled atomic clause is extended to allow a procedure call, in addition to an action,

where m is a monitor name, and pn is a procedure name.

$$\begin{aligned} &\mathit{Body} ::= \\ &\quad \mathit{Action} \\ &\quad \mid [v :=] m.pn(a_1, \ldots, a_k) \end{aligned}$$

Monitors is a collection of monitors over monitor names,

Definition 3.13. $Monitors = (CName \rightarrow Monitor)$

A monitor has a set of variables and some procedures,

Definition 3.14. $Monitor = \mathbb{P}(Var) \times Procedures$

Procedures is a collection defined by a function over procedure names,

Definition 3.15. $Procedures = (PdName \rightarrow Procedure)$

A procedure consists of local variable definitions (the formal parameters) guards and actions and may specify a return type.

Definition 3.16. $Procedure = LVar \times Guard \times Action \times T$

We define a TLA mapping for the new clause. We ensure the type of the return variable matches the assigned variable in a static check. We impose restrictions on A, so $f_1, \ldots, f_k$ can only appear in guards and expressions; and **return** only appears on the LHS of an assignment.

Definition 3.17. $< \ l1 : \ \triangleleft \ g_{\mathrm{c}} \rightarrow v := m.pn(a_1, \ldots, a_k) \ \triangleright \ , \ l2, \ P >^{TLA}$
$\triangleq$
$l1_{\mathrm{P}} \ =$
WHEN $P_{\mathrm{pc}} = l1 \wedge g_{\mathrm{p}}[f_1, \ldots, f_k \backslash a_1, \ldots, a_k] \wedge g_{\mathrm{c}}$
THEN $A[f_1, \ldots, f_k \backslash a_1, \ldots, a_k][\mathbf{return} \backslash v] \parallel P_{\mathrm{pc}} := l2$
END

where procedure pn of monitor m is defined by $m.pn(f_1, \ldots, f_k) = g_{\mathrm{p}} \rightarrow A$

In the mapping we use substitution; formal parameters are substituted for actual parameters in the guard and action, and the **return** variable is substituted by the assigned variable on the LHS of the call. We show a small example of substitution where a variable of the caller, v, is assigned the value returned by a procedure call, pn. We assume the monitor has some variables, x and r. We define the procedure, $pn(Integer\ z)\{x := z \parallel return := r\}$, and call $v := m.pn(y)$. Then substitution is as follows, $(x := z \parallel return := r)[z \backslash y][return \backslash v] = (x := y \parallel v := r)$. Substitution for

guards is similar to that for actions. It is also worth highlighting the point, at this juncture, that the procedure definitions of a monitor m, $m.pn(f_1, \ldots, f_k) = g_\mathrm{p} \rightarrow A$ are not translated directly into Event-B. Procedure definitions are used in the translation of procedure calls as described above, and the resulting event may contain additional guards from the containing non-atomic clause (such as those derived from branch conditions), or assignment of a return value.

3.6 Review of the Chapter

In this chapter we introduced the first aspects of our main contribution. We introduced the notion of processes to OCB in order to model entities performing some tasks. Such a notion does not exist in the Event-B language, but we can model processes in Event-B. By relating OCB and Event-B we are able to define semantics for OCB using an Event-B model. We introduced the notion of ordered, interleaving non-atomic operations for processes. Event-B events can be used to model the activities performed by interleaving processes, however the Event-B language itself does not provide a method of imposing ordering of occurrence of events, however the order in which events can occur can be dictated by appropriate choice of guards. We introduced the notion of operations which are able to interleave and make use of sequences (using the semi-colon operator) of labelled atomic clauses. The use of labelled atomic clauses simplifies the reasoning process by using a simple notion of atomicity. During translation each labelled atomic clause maps to an individual event. In the presentation we introduced the syntax of atomic and non-atomic constructs in terms of the Guarded Command Language, and we have shown how the constructs are related to Event-B; that is they are given Event-B semantics.

We wish to alleviate the developer from the burden of defining the ordering guards explicitly, by using sequences of labelled atomic clauses to describe the ordered executions of processes. The labels map to program counter values and the translator automatically adds the appropriate program counters to event guards, and program counter updates to actions. The sequences of labelled atomic clauses described above are facilitated by non-atomic constructs; we have already mentioned the sequence clauses which make use of the semi-colon sequence operator. In addition we add further expressivity to OCB with non-atomic clauses to facilitate looping, and branching behaviour. Once again such looping and branching behaviour is not part of Event-B language and we wish to provide these higher level constructs as part of our OCB language.

We introduced monitors and atomic procedure calls to OCB in order to model entities sharing data between the processes; where monitors provide the mutual exclusion mechanism. An advantage of using a monitor abstraction in OCB is that it alleviates

the developer from the burden of reasoning about locking and implementing conditional waiting.

Chapter 4

The OCB Language Part 2 - Object-Oriented Features

In this chapter we introduce OCB's object-oriented features, and then describe the syntactic sugar that is used in the textual notation. Following this we show an example OCB specification with its Event-B representation, the implementation in Java, and finally discuss the Java translation rules. This chapter is a continuation of the previous chapter which expands upon our paper [54].

4.1 Mapping Object-Oriented Features to Event-B

In the previous chapter we introduced processes and monitors, and until now there has only been one process or monitor associated with a given monitor or process name. We wish to extend the system to allow the use of their definitions as templates for instantiation of objects; we refer to the process and monitor definitions as class definitions. In order to facilitate instantiation we introduce constructor procedures with the reserved name, **create**; new instances are constructed by processes invoking the **create** procedure. Each monitor and process class can have a constructor procedure where initialisation of variables takes place. Initialisation of a class' Integer and Boolean typed attributes in the constructor is mandatory, since we map OCB attributes of these types to total functions. In Java the initialisation of primitive types is as follows, **int**s are initialized to zero and **boolean** types are initialized to **false**. In our work we could have adopted the same initial values, but we decided to ensure OCB attributes are initialized in the constructor, and perform a static check at the time of translation to ensure that this has been done. Therefore the initial value of an attribute has to be explicitly specified, which should be of benefit in future work, the default value may be different when extending the approach to other target platforms. When declaring attributes that are class types the initial value may be null, a new child object may be constructed at some

later time, but attribute declaration is always related to the parent instantiation. We therefore map attributes that refer to an instance using a partial function, this means their initialisation in the constructor is optional. Actual parameters, a, supplied to constructors may be used to initialize attributes, by substitution of formal parameters, f.

A system is modelled as a special class called $MainClass$, its non-atomic clause corresponds to the Java *main* method - the entry point for execution in the implementation. Since the main class is a kind of class we use a name from $CName$,

Definition 4.1. $MainClass ::= CName$
$ProcessClass^*$
$MonitorClass^*$
Var^*
$NonAtomic$

The main class has a name, a non-atomic clause, an optional number of process class and monitor class definitions, and some attribute declarations Var. Our approach uses techniques introduced in UML-B [140], to model object-oriented features. We adopt the UML-B style of modelling classes and object instantiation; to which we add processes, non-atomic operations, program counters, and monitors. As in UML-B, for each class C we add a variable $C_{\text{inst}} \subseteq C$ to represent the current set of instances of C. Each attribute declaration $v \in T$ of class C maps to a variable with the same name in Event-B, which is typed $v \in C_{\text{inst}} \rightarrow T$. A process' non-atomic operations contain labelled atomic clauses which map to events, and for each process class P in the system we model the flow of execution, from one labelled clause to the next, using a program counter variable P_{pc}. This is typed $P_{\text{pc}} \in P_{\text{inst}} \rightarrow P_{\text{Label}}$. Program counter values in P_{Label} correspond to the labels of the atomic clauses, plus the terminating state. To create an instance of a class we invoke its constructor by calling its **create** procedure. We modify the syntax of *Body* of Chapter 3 to accommodate a constructor call to instantiate a class C with actual parameters $a_1, \ldots, a_k$.

$Body ::=$
$Action$
$\mid [v :=] m.pn(a_1, \ldots, a_k)$ (procedure call)
$\mid v := C.\textbf{create}(a_1, \ldots, a_k)$ (constructor call)

We will also see later that special treatment is required when translating the attribute references used in the bodies of atomic clauses. It is assumed that when we instantiate a process its processing can begin immediately; that is, in the implementation the threads are started immediately following creation. We now introduce definitions for

ProcessClass and *MonitorClass*,

$$ProcessClass \ ::= \ CName \ Var^{+} \ NonAtomic \ Constructor$$

$$MonitorClass \ ::= \ CName \ Var^{*} \ Procedure^{+} \ Constructor$$

Each process class has a unique name from $CName$ and one or more unique (within the class) attributes Var. Attributes of Var can be Integer, Boolean, array, or class types. Similarly each monitor class has a unique name $CName$ and zero or more attributes. Monitor class attributes can be Integer, Boolean or class types. But it should be noted that monitors may not call procedures of other monitors directly, however monitor objects can be passed to a process as a return value of a procedure call, then the process can invoke a procedure on the monitor. The syntax of a process non-atomic clause, and monitor procedure is as previously described in Chapter 3; the constructor procedure syntax is as follows,

Definition 4.2. $Constructor \ ::= \ LVar^{*} \ Action \ Type$

The constructor consists of zero or more formal parameter declarations $LVar$, $Action$ which describes the initialisation actions, and the type $Type$ being constructed - which is implicitly the class in which the create procedure definition is contained. The TNA mapping function for OCB's object oriented non-atomic clauses remains the same, as does the TLA mapping function.

When an attribute is used in an OCB clause its use is with respect to the class in which it is declared, the specification is general in the sense that when we describe an action in an OCB clause it can be applied to any instance. However, when we map to the Event-B model we need to model the attribute with respect to a particular instance of the class, which we represent in an event as a parameter. The instance represented by the parameter may be the owner of the labelled clause; the target of a procedure call; or a new instance, in the case of constructor initialisations. The mapping of an attribute reference occurs in both actions and guards of the Event-B model. If an OCB class C has a declaration introducing v, and a clause referencing attribute v, then the attribute declaration maps to an Event-B variable v as in UML-B. To model the attribute reference in the event we introduce an event parameter s to represent an instance of class C; then in the event guard and action $v(s)$ is the corresponding representation of the attribute in Event-B. To map the use of an attribute in OCB to its variable representation in Event-B we apply the function, TV. This function takes a guard, action or expression as a parameter, and maps it to the corresponding Event-B representation. $VarName$ is the set of attribute names belonging to the class being referred to, and $EventBLVar$ is the name of the Event-B parameter representing the instance. The type of TV is defined as follows,

Definition 4.3. $TV \in (Guard \cup Action \cup E) \times \mathbb{P}(VarName) \times EventBLVar \rightarrow (Guard \cup Action \cup E)$

The variable renaming function TV may need to be applied a number of times to a clause; we apply it once for each class with attributes mapped to the event guard or action. TV is applied to an action a, expression E, or guard g as shown in Table 4.1. We use the notation *gbOp* to refer to the OCB binary operators relating to the guards, *guOp* to refer to unary operators relating to guards. Operators *abOp* and *auOp* refer to binary and unary arithmetic operators.

Action a	$< a,\ vn,\ s >^{TV}$
$a_1 \parallel a_2$	$< a_1,\ vn,\ s >^{TV} \parallel < a_2,\ vn,\ s >^{TV}$
$v := E$	$< v,\ vn,\ s >^{TV} := < E,\ vn,\ s^{TV}$
Guard g	$< g,\ vn,\ s >^{TV}$
g_1 gbOp g_2	$< g_1,\ vn,\ s >^{TV}$ gbOp $< g_2,\ vn,\ s >^{TV}$
guOp g	guOp $< g,\ vn,\ s >^{TV}$
BooleanLiteral	*BooleanLiteral*
Expression E	$< E,\ vn,\ s >^{TV}$
IntegerLiteral	*IntegerLiteral*
BooleanLiteral	*BooleanLiteral*
v	$v(s)$ where $v \in vn$
	v where $v \notin vn$
e_1 abOp e_2	$< e_1,\ vn,\ s >^{TV}$ abOp $< e_2,\ vn,\ s >^{TV}$
auOp e	auOp $< e,\ vn,\ s >^{TV}$

TABLE 4.1: Variable Renaming with TV

We show an example mapping with a attribute v, used in a labelled assignment $l1 : v := v+1$, with a calling instance s. The mapping using TV where $vn = \{v\}$ follows,

$$< v := v + 1,\ vn,\ s >^{TV} = v(s) := v(s) + 1$$

The effect of the application of TV is that wherever a variable name appears in the clause and also in the set of names vn a function application is created with respect to s. Other literals and variables not in vn, and operators, are unchanged.

In subsequent definitions we use the following Event-B syntax for a guarded action with parameters, **ANY** L **WHERE** G **THEN** S **END**. L is a list of parameters, G is a guarding predicate, and the body S contains some assignment actions. The following notation is used for any process class P, P_{pc}(s), is the program counter for process instances of class P. P_{inst} is the set of current instances of class P. P_{set} is the set of potential instances of class P. The new definition of TLA for a labelled atomic clause follows where the clause is defined in class P,

Definition 4.4. $< l1 :\ \triangleleft\ g \rightarrow A\ \triangleright\ ,\ l2,\ P >^{TLA}$
$\triangleq$
$l1_{\mathrm{P}}\ =$
ANY s
WHERE $s \in P_{\mathrm{inst}} \wedge P_{\mathrm{pc}}(s) = l1 \wedge < g,\ vn,\ s >^{TV}$
THEN $< A,\ vn,\ s >^{TV} \| P_{\mathrm{pc}}(s) := l2$
END

where vn is the set of variable names of class P.

The definition of a labelled constructor clause follows, where P creates a new instance of process class Q,

Definition 4.5. $< l1 :\ \triangleleft\ g_{\mathrm{c}} \rightarrow v := Q.\mathbf{create}(a_1, \ldots, a_k)\ \triangleright\ ,\ l2,\ P >^{TLA}$
$\triangleq$
$l1_{\mathrm{P}}\ =$
ANY new, s
WHERE $s \in P_{\mathrm{inst}} \wedge P_{\mathrm{pc}}(s) = l1 \wedge new \in Q_{\mathrm{set}} \setminus Q_{\mathrm{inst}} \wedge$
$< g_{\mathrm{c}},\ vq,\ s >^{TV}$
THEN $< A',\ vp,\ new >^{TV} \| P_{\mathrm{pc}}(s) := l2 \|$
$Q_{\mathrm{inst}} := Q_{\mathrm{inst}} \cup \{new\} \| v(s) := new \| Q_{\mathrm{pc}}(new) := sLabel(na)$
END

For this definition we state that na is the non-atomic clause of class Q, and vq and vp are sets of variable names, of the caller, and new instance respectively. We assume the constructor procedure body A is defined in the following way,

$$Q.create(f_1, \ldots, f_k) = A$$

A' is the action A with occurrences of formal parameters substituted by actual parameters. The actual parameters may be values; or mapped attributes which are resolved with respect to the calling instance. We define A' as,

$$A' \triangleq A[fn_1, \ldots, fn_k \backslash < a_1,\ vq,\ s >^{TV}, \ldots, < a_k,\ vq,\ s >^{TV}] \tag{4.1}$$

here $fn_1, \ldots, fn_k$ are the names of the formal parameters.

A similar mapping exists for monitor class instantiation, but excludes setting of a program counter; monitors do not have program counters since they play a passive role in the system. The definition of a labelled constructor clause follows, where P creates a new instance of monitor class M,

Definition 4.6. $< l1 :\ \triangleleft\ g_{\mathrm{c}} \rightarrow v := M.\mathbf{create}(a_1, \ldots, a_k)\ \triangleright\ ,\ l2,\ P >^{TLA}$
$\triangleq$
$l1_{\mathrm{P}}\ =$
ANY new, s
WHERE $s \in P_{\mathrm{inst}} \wedge P_{\mathrm{pc}}(s) = l1 \wedge new \in M_{\mathrm{set}} \setminus M_{\mathrm{inst}} \wedge$
$< g_{\mathrm{c}},\ vq,\ s >^{TV}$
THEN $< A',\ vp,\ new >^{TV} \| P_{\mathrm{pc}}(s) := l2 \|$
$M_{\mathrm{inst}} := M_{\mathrm{inst}} \cup \{new\} \| v(s) := new$
END

For this definition we assume that vq and vp are sets of variable names, of the caller, and new instance respectively. Assume that the constructor procedure is defined in the following way, with body A

$$M.create(f_1, \ldots, f_k) = A$$

A' is the action A with occurrences of formal parameters substituted by actual parameters. The actual parameters may be values; or mapped attributes which are resolved with respect to the calling instance. We define A' in the following way,

$$A' \triangleq A[fn_1, \ldots, fn_k \backslash < a_1,\ vq,\ s >^{TV}, \ldots, < a_k,\ vq,\ s >^{TV}]$$

here $fn_1, \ldots, fn_k$ are the names of the formal parameters.

We now look at the definition of TLA for monitor procedure calls. We define a translation for a procedure call $m.pn(a_1, \ldots, a_n)$, where pn is the procedure name, and m is the target instance which is a variable belonging to instance s. We assume that the procedure call, pn, is defined in a monitor class m in the following way,

$$m.pn(a_1, \ldots, a_k) = g_{\mathrm{p}} \rightarrow A$$

and $j = m(s)$ gives an event parameter referring to the monitor instance being called. We will perform a static check to ensure the return type of the procedure matches the variable being assigned to, and we prohibit use of the **return** variable in g_{p} and expressions (RHS of assignments) in A. There are two events resulting from this mapping, the second handles the case where the target m does not exist for some reason in the caller s. In the implementation an exception is thrown and the process terminates, we model this by assigning the terminating label tp to the program counter.

Definition 4.7. $< l1 :\ \triangleleft\ g_{\mathrm{c}} \rightarrow v := m.pn(a_1, \ldots, a_n)\ \triangleright\ ,\ l2,\ P >^{TLA}$
$\triangleq$
$l1_{\mathrm{P}}\ =$

ANY s, j
WHERE $s \in P_{\text{inst}} \wedge P_{\text{pc}}(s) = l1 \wedge s \in dom(m) \wedge j = m(s) \wedge$
$< g_{\text{p}}{}',\ vj,\ j >^{TV} \wedge < g_{\text{c}} >,\ vq,\ s >^{TV}$
THEN $< A',\ vj,\ j >^{TV} \| P_{\text{pc}}(s) := l2$
END

$l1\_null_{\text{P}}$ =
ANY s
WHERE $s \in P_{\text{inst}} \wedge \text{s} \in \text{dom}(P_{\text{pc}}) \wedge P_{\text{pc}}(s) = l1 \wedge s \notin dom(m)\ \wedge$
$< g_{\text{c}},\ vq,\ s >^{TV}$
THEN $P_{\text{pc}}(s) := tp$
END

where vq and vj are sets of variable names of the caller, and monitor instance respectively; and the procedure call $m.pn$ is equivalent to the guarded action, In the definition $g_{\text{p}}{}'$ is a procedure guard with some (but not all) substitutions applied. We apply substitutions in two stages, the first stage shown below is used to substitute actual parameters for formal parameters in the guard; these actual parameters need to be mapped to variables associated with the calling instance using TV before substitution. The substitutions related to monitor class attributes are then performed by applying TV to $g_{\text{p}}{}'$ (seen above). The first stage substitutions are follows,

$$g_{\text{p}}{}' = \text{g}_{\text{p}}[fn_1, \ldots, fn_k \backslash < a_1,\ vq,\ s >^{TV}, \ldots, < a_k,\ vq,\ s >^{TV}]$$

where $fn_1, \ldots, fn_k$ are the names of the formal parameters.

The action clause A' is an action with some of the substitutions applied. Again there are two stages, and in the first stage the formal parameters of the called class are substituted for actual parameters with TV applied as follows, then TV is applied to A' (seen above) to map monitor class attributes. We define A' in the following way,

$$A' \triangleq A[fn_1, \ldots, fn_k \backslash < a_1,\ vq,\ s >^{TV}, \ldots, < a_k,\ vq,\ s >^{TV}][\textbf{return} \backslash v]$$

where $fn_1, \ldots, fn_k$ are the names of the formal parameters.

4.1.1 OCB Arrays

Until now we have assumed that OCB attribute types are restricted to simple Integer, Boolean and class types. We will however find it useful to have a representation of an array of elements of these types. The array we define is similar to that of Java in that it is zero-based and of fixed length. To facilitate array declarations we introduce OCB

notation of the form $T[n]\ v$, where T is either Integer, Boolean or a class name; n is the array capacity; and v is an attribute name. Array accesses are performed using index notation $v[i]$, where i is typed as $i \in 0 \ldots n-1$. For any class in C_{inst} with attribute named v we map to an Event-B variable with the same name. The type definition in the invariant clause depends, however, on the element type; we define T_p as the primitive types Integer and Boolean, and T_c as any class type denoted by class name.

$$v \in C_{inst} \rightarrow (0 \,..\, n-1) \rightarrow T_p$$

The definition associated with primitive types gives rise to a function where the range is a total function from indices to values. We assume that Integer array elements are initialized to zero at the time of construction (of the class), and Boolean array elements are initialized to **FALSE**. These values correspond to the Java implementation where integer elements are initialized to zero, and boolean elements to **false**. The type definition for an array of objects follows,

$$v \in C_{inst} \rightarrow (0 \,..\, n-1) \nrightarrow T_c$$

In the constructor event of the class which defines the array attribute, initialisation of an Integer array is performed by the following action, $v(new) := \lambda.i(i \in 0..n-1|0)$; where n is the capacity of the array, and new is the new object being initialized. A similar initialisation is used for boolean attributes.

An array of objects initially contains no elements so the range is a partial function. An array may be used to store both process and monitor objects; but we currently have no need to store process objects in arrays, since we have no requirement to refer to them after instantiation. The current OCB approach creates a process instance and its run operation is executed. We never refer to it externally since process classes are not shared. When accessing an array element that stores a monitor object it is possible to call one of the object's procedures. Given a $MonitorClassM$ and an array attribute $M[n]\ v$ containing M type objects, we may reasonably expect to locate a monitor object in the array and call a procedure. We use the notation $v[i].op()$ to do this, where i is an integer index, and op is a procedure of M. Now it may be the case that the call is not possible because their is no monitor object at that index. In effect $v[i]$ would be null in the Java implementation, or in Event-B for an instance $s \in C_{inst}$, $v(s)(i)$ would be undefined. We can ensure that any such call in a clause labelled *l1* is well-defined by adding an invariant clause as follows.

$$\forall s \cdot s \in C_{inst} \wedge C_{pc}(s) = l1 \Rightarrow s \in dom(v) \ \wedge \ i \in \ dom(v(s))$$

To translate calls of this type we simply use the TLA translation function 4.7 with $m = v[i]$, then the event that handles a null target also accommodates a null array

element. So we have,

$$< l1 : \ \lhd \ g_{\mathrm{c}} \rightarrow v := v[i].pn(a_1, \ldots, a_n) \ \rhd \ , \ l2, \ P >^{TLA}$$
$$=$$
$$< l1 : \ \lhd \ g_{\mathrm{c}} \rightarrow v := m.pn(a_1, \ldots, a_n) \rhd, \ l2, \ P >^{TLA}$$

We now consider the issue of array bound checking. The indices that we use for arrays of size n should be in the range $0 \,..\, n-1$. We do not explicitly check the array index is in range; instead an array access or assignment gives rise to a proof obligation which we are required to discharge. Discharging this proof obligation will ensure that the index is in range. In the following example we show an attempt to assign to an out of range index, which gives rise to a proof obligation that cannot be discharged. Assume that the array variable $Integer[5]$ arr is declared in a class P. An index value of 5 is out of range since $5 \notin 0 \,..\, 4$. The erroneous labelled atomic clause is,

$$l1 : \ arr[5] := 99$$

The labelled atomic clause maps to the following event action,

$$arr(self) := arr(self) \lhd\!\!+ \{5 \mapsto 99\}$$

The RODIN tool produces the following proof obligation, which we will be unable to discharge; since clearly $5 \notin 0 \,..\, 4$,

$$arr \lhd\!\!+ \{self \mapsto arr(self) \lhd\!\!+ \{5 \mapsto 99\}\} \in P \rightarrow (0 \,..\, 4 \rightarrow Z)$$

Next we come to the issue of referring to array elements on the RHS of assignment expressions and in predicates. A simple array reference $v[i]$ such as that used in the assignment $x := v[i]$ is translated using the previously introduced TV function. We apply TV to resolve any attribute names in an expression, this may include the array index i. The application of TV to an array reference rewrites the array name with respect to the class it is in, and will also rewrite the array index if it is an attribute name. So where vc is a set of variables of the owning process/monitor class, and s is the owning process, we rewrite the array expression as follows,

$$< v[i], \ vc, \ s >^{TV} \ = \ v(s)(< i, \ vc, \ s >^{TV})$$

In OCB we may have an assignment $x := v[i]$, which assigns a value $v[i]$ to an array element x. In this case we need to apply TV to both left and right hand sides of the

assignment as follows,

$$
\begin{aligned}
& < x := v[i],\ vc,\ s >^{TV} \\
& = \\
& < x,\ vc,\ s >^{TV} := v(s)(< i,\ vc,\ s >^{TV}) \\
& = \\
& x(s) := v(s)(< i,\ vc,\ s >^{TV})
\end{aligned}
$$

At first sight we may consider simplifying the right hand side further to match the left hand side, but we do not do this since $< i,\ vc,\ s >^{TV}$ may result in an integer literal index.

We now discuss OCB atomic actions involving array updates, that is where an array value appears on the left hand side of an assignment expression. In OCB we can write $v[i] := x$, for some integer array v we can assign an integer value x to some element indexed by i. Applying TV to the expression we obtain,

$$
\begin{aligned}
& < v[i] := x,\ vc,\ s >^{TV} \\
& \triangleq \\
& v(s) := v(s) \lhd \{< i,\ vc,\ s >^{TV} \mapsto < x,\ vc,\ s >^{TV}\}
\end{aligned}
$$

where $\lhd$ is function override operator. An action can involve a number of terms with v on the LHS of an assignment, such as $v[i] := x \parallel \ldots \parallel v[j] := y$. We can generalize this approach as long as the indices of the array are unique. In any action, we collect each assignment expression with an array reference v appearing on the LHS, and apply TV to create a single update which uses relational override as follows,

$$
\begin{aligned}
& < v[i] := x \parallel \ldots \parallel v[j] := y,\ vc,\ s >^{TV} \\
& \triangleq \\
& v(s) := v(s) \lhd \{< i,\ vc,\ s >^{TV} \mapsto < x,\ vc\ s >^{TV}, \\
& \qquad \ldots, < j,\ vc,\ s >^{TV} \mapsto (, < y,\ vc,\ s >^{TV}\}
\end{aligned}
$$

It then remains to prove that the indices are indeed unique, and in bounds. Bound checking is performed by proof, the access shown above will give rise to a proof obligation of the form $i \in dom(v(self))$ which should be discharged.

4.2 Syntactic Sugar for Specification

The guarded command language has served as a useful notation for defining the mapping to Event-B. We can however define syntactic sugar to provide a notation which

is more familiar to implementers of object-oriented systems. We provide the following programmatic style notation. Firstly we introduce an if style choice construct.

Definition 4.8. $l1$: **if**(g_1) **then** b_1 **andthen** na_1 **endif**
elseif(g_2) **then** b_2 **andthen** na_2 **endelseif**...
else b_n **andthen** na_n **endelse**

$\triangleq$

$$
\begin{array}{l}
l1: \ \triangleleft\ g_1 \rightarrow b_1\ \triangleright\ ; na_1 \\
[]l1: \ \triangleleft\ \neg g_1 \wedge g_2 \rightarrow b_2\ \triangleright\ ; na_2 \ldots \\
[]l1: \ \triangleleft\ \neg g_1 \wedge \neg g_2 \wedge \ldots \wedge \neg g_n \rightarrow b_n\ \triangleright\ ; na_n
\end{array}
$$

The looping construct is presented in the form of a *while* loop,

Definition 4.9. $l1$: **while**(g) **then** b **andthen** na **endwhile**

$\triangleq$

$$do\ l1: \ \triangleleft\ g \rightarrow b\ \triangleright\ ; na\ od$$

We use a **when** clause to guard monitor procedures,

Definition 4.10. **when**$(g_{\mathrm{p}})\{A\}$

$\triangleq$

$$\triangleleft\ g_{\mathrm{p}} \rightarrow A\ \triangleright$$

The mapping to Event-B is intended to be fully automated, and we give details of the tool support in Chapter 5. The sugared form of specification also provides for a simpler mapping and automatic translation to Java. This is described later in Section 4.4.

4.3 An Example Mapping to Event-B

In this section we will show how a simple OCB specification gives rise to an Event-B model. Figure 4.1 shows an OCB specification with two *ProcessClass*, a *MonitorClass*, and *MainClass*, definitions. The *MonitorClass* called *Shared* has two attributes, *Integer v* and *Integer tally*. The shared *MonitorClass* stores a value in v. The constructor procedure initialises these attributes. The *MonitorClass* has procedures to set and get the value of v, $setVal$ and $getVal$, and keeps a tally of read attempts in *tally*. The *MonitorClass* is shared between two process instances, *Get* and *Put*. The *Get* process stores a reference to the *Shared* instance in its *shared* attribute; the *Shared* instance is passed to the process, and the shared attribute is initialised, in its constructor procedure. The *run* operation's $l1$ clause specifies looping behaviour. The loop calls the *Shared* instance's $getVal$ procedure to obtain the value of v and assigns the value to i. The clause $l2$ then updates the *Get* class' counter attribute which is used to guard the loop. The *Put* process instance provides new values for the shared class' v attribute. It does

```
ProcessClass Get{
 Shared shared , Integer i, Integer count
 Procedure create(Shared shd){ shared := shd ||
      i := 0 || count := 0 }
 Operation run(){
    l1: while(count < 100) do i := shared.getVal()
        andthen l2: count := count + 1 endwhile
 }
}

ProcessClass Put{
 Shared shared, Integer val, Integer count
 Procedure create(Shared shd){ shared := shd ||
     val := 0 || count := 0 }
 Operation run(){
    k1: while(count < 100) do val := count mod 5
        andthen k2: shared.setVal(val) ;
                    k3: count := count + 1 endwhile
 }
}

MonitorClass Shared{
 Integer v, Integer tally
 Procedure create(){ v := 1 || tally := 0 }
 Procedure getVal(){
    when(v > 0){ tally := tally + 1 || return := v}}:Integer
 Procedure setVal(Integer vl){ v := vl }
}

MainClass Main{
 Put pu, Get ge, Shared sh
 Operation main(){
    m1: sh := Shared.create();
    m2: pu := Put.create(sh) ;
    m3: ge := Get.create(sh)
 }
}
```

FIGURE 4.1: An Example OCB Specification

this by calling the *Shared* instance's $setVal$ procedure in clause $k2$ which resides within the loop guarded by the condition of clause $k1$. The $MainClass$ provides the entry point for execution. The attributes are used to keep track of the instances; the *Shared* instance is created in clause $m1$ and passed to the process constructors in clauses $m2$ and $m3$. The $getVal$ procedure demonstrates the use of a conditional waiting clause; a value is only returned by the procedure if it is greater than zero and is caused to wait otherwise.

In the examples that follow we use a sanitized mapping for clarity. This is mainly achieved by shortening names, for example, the frequently used parameter $self$ appears as s; at a later stage we will present actual translations. Where a variable name is used in more than one class we have added a subscript to identify which class they belong to. The attributes of the OCB classes give rise to variables in the Event-B model, and these are typed in the invariant as follows. We make use of $Main_{\text{set}}$ and Put_{set} which are carrier sets defined in the associated context.

$$
\begin{aligned}
Main_{\text{inst}} &\in \mathbb{P}(Main_{\text{set}}) \\
Main_{\text{pc}} &\in Main_{\text{inst}} \pfun Main_{\text{ctr}} \\
Put_{\text{inst}} &\in \mathbb{P}(Put_{\text{set}}) \\
Put_{\text{pc}} &\in Put_{\text{inst}} \pfun Put_{\text{ctr}} \\
Get_{\text{inst}} &\in \mathbb{P}(Get_{\text{set}}) \\
Get_{\text{pc}} &\in Get_{\text{inst}} \pfun Get_{\text{ctr}} \\
shared_{\text{put}} &\in Put_{\text{inst}} \pfun Shared_{\text{inst}} \\
shared_{\text{get}} &\in Get_{\text{inst}} \pfun Shared_{\text{inst}} \\
sh &\in Main_{\text{inst}} \pfun Shared_{\text{inst}} \\
val &\in Put_{\text{inst}} \to \mathbb{Z} \\
count_{\text{put}} &\in Put_{\text{inst}} \to \mathbb{Z} \\
count_{\text{get}} &\in Get_{\text{inst}} \to \mathbb{Z} \\
pu &\in Main_{\text{inst}} \pfun Put_{\text{inst}} \\
ge &\in Main_{\text{inst}} \pfun Get_{\text{inst}} \\
i &\in Get_{\text{inst}} \to \mathbb{Z}
\end{aligned}
$$

A process' program counter can take values mapped from its labels, plus the terminating label $main_{\text{t}}$. The labels map to constant names which form an enumerated set of available values. The $MainClass$ program counter $Main_{\text{pc}}$ is a mapping from instances of main to an enumerated set of program counters defined in $Main_{\text{ctr}}$; similarly the program counters for the instances of the Get class are defined in Get_{ctr}, and for the instances of Put we have Put_{ctr}. The enumerated sets of program counters are axioms, constructed by the translator, in the context as follows,

$$
\begin{aligned}
Main_{\text{ctr}} &= \{m1, m2, m3, main_{\text{t}}\} \\
Get_{\text{ctr}} &= \{l1, l2, get_{\text{t}}\} \\
Put_{\text{ctr}} &= \{k1, k2, k3, put_{\text{t}}\}
\end{aligned}
$$

In the following example we show the mapping of a **create** call in the clause labelled $m2$ of $MainClass\ Main$. This maps to an event $m2_{\text{Main}}$ in which a new Put process instance is created and initialized, and the program counter is updated with the next

label in the sequence.

$m2_{\text{Main}}$ =
ANY new, s
WHERE $s \in Main_{\text{inst}} \wedge s \in dom(Main_{\text{pc}}) \wedge Main_{\text{pc}}(s) = m2 \wedge$
$new \in Put_{\text{set}} \setminus Put_{\text{inst}} \wedge \text{s} \in \text{dom(sh)}$
THEN $shared_{\text{put}}(new) := sh(s) \parallel val(new) := 0 \parallel count_{\text{put}}(new) := 0 \parallel$
$Main_{\text{pc}}(s) := m3 \parallel Put_{\text{inst}} := Put_{\text{inst}} \cup \{new\} \parallel pu(s) := new \parallel$
$Put_{\text{pc}}(new) := k1$
END

The event action is simply constructed as per definition 4.5. We discuss the substitution of formal parameters and variable renaming since it is one of the more complex aspects of the mapping. We have a formal parameter shd replaced by sh of $Main$ (but renaming of the actual parameter sh takes place first). So using equation 4.1 and $vq = \{pu, ge, sh\}$ we have,

$$\begin{aligned} A' =& (shared_{\text{put}} := shd \parallel val := 0 \parallel count_{\text{put}} := 0)[shd \backslash < sh,\ vq,\ s >^{TV}] \\ =& \text{ “substitution, } < sh,\ vq,\ s >^{TV} = sh(s)\text{”} \\ & (shared_{\text{put}} := sh(s) \parallel val := 0 \parallel count_{\text{put}} := 0) \end{aligned}$$

Then the variables $vp = \{shared_{\text{put}}, val, count_{\text{put}}\}$ associated with the newly constructed object are renamed.

$$\begin{aligned} & < shared_{\text{put}} := sh(s) \parallel val := 0 \parallel count_{\text{put}} := 0\ ,\ vp,\ new >^{TV} \\ =\ & < A',\ vp,\ new >^{TV} \\ & shared_{\text{put}}(new) := sh(s) \parallel val(new) := 0 \parallel count_{\text{put}}(new) := 0 \end{aligned}$$

Similar events model the construction of the *Get* process class and the *Shared* monitor class instances. We now consider the translation of the *run* operation of *ProcessClass Put*. This gives rise to five events relating to: $k1$ with a true guard; $k1$ with a false guard; $k2$ with a procedure call, $k2$ null handler (the target does not exist) and $k3$ counter update. We now show the event $k1_{\text{Put}}$, the true branch that arises from the clause labelled $k1$

containing a loop condition.

$$
\begin{array}{l}
k1\_while_{\mathrm{Put}} \ = \\
\textbf{ANY } s \\
\textbf{WHERE } s \in Put_{\mathrm{inst}} \wedge s \in dom(Put_{\mathrm{pc}}) \wedge Put_{\mathrm{pc}}(s) = k1 \wedge \\
\quad count(s) < 100 \\
\textbf{THEN } val(s) := count \ mod \ 5 \parallel Put_{\mathrm{pc}}(s) := k2 \\
\textbf{END}
\end{array}
$$

The **WHERE** predicate ensure that s is in the set of instances, and that the program counter is $k1$. $s \in dom(Put_{\mathrm{pc}})$ simply ensures the succeeding predicate element is well defined. $count(s) < 100$ arises directly from the loop condition. The event $k1\_false_{\mathrm{Put}}$ that arises due to the false guard follows,

$$
\begin{array}{l}
k1\_false_{\mathrm{Put}} \ = \\
\textbf{ANY } s \\
\textbf{WHERE } s \in Put_{\mathrm{inst}} \wedge s \in dom(Put_{\mathrm{pc}}) \wedge Put_{\mathrm{pc}}(s) = k1 \wedge \\
\quad \neg(count(s) < 100) \\
\textbf{THEN } Put_{\mathrm{pc}}(s) := put_{\mathrm{t}} \\
\textbf{END}
\end{array}
$$

This event contains the negated loop condition and sets the program counter to the value of the terminating state, since there is no other successor state following loop termination. The third event, $k2_{\mathrm{Put}}$, arises from the second clause, labelled $k2$, with a procedure call,

$$
\begin{array}{l}
k2_{\mathrm{Put}} \ = \\
\textbf{ANY } s, m \\
\textbf{WHERE } s \in Put_{\mathrm{inst}} \wedge s \in dom(Put_{\mathrm{pc}}) \wedge Put_{\mathrm{pc}}(s) = k2 \wedge \\
\quad s \in dom(shared_{\mathrm{put}}) \wedge m = shared_{\mathrm{put}}(s) \\
\textbf{THEN } v(m) := val(s) \parallel Put_{\mathrm{pc}}(s) := k3 \\
\textbf{END}
\end{array}
$$

The procedure call of $k2$ is expanded in-line, where the update of the procedure body can be seen with the actual parameter substituted for the formal parameter. For cases where the call is to a null target we provide the following event which terminates the

calling process.

$$
\begin{array}{l}
k2\_isNull_{\mathrm{Put}} \ = \\
\textbf{ANY } s \\
\textbf{WHERE } s \in Put_{\mathrm{inst}} \wedge s \in dom(Put_{\mathrm{pc}}) \wedge Put_{\mathrm{pc}}(s) = k2 \wedge \\
\quad s \notin dom(shared_{\mathrm{put}}) \\
\textbf{THEN } Put_{\mathrm{pc}}(s) := put_{\mathrm{t}} \\
\textbf{END}
\end{array}
$$

The fifth event, $k3_{\mathrm{Put}}$, arising from the third clause labelled $k3$ updates *count* which is used in the loop condition,

$$
\begin{array}{l}
k3_{\mathrm{Put}} \ = \\
\textbf{ANY } s \\
\textbf{WHERE } s \in Put_{\mathrm{inst}} \wedge s \in dom(Put_{\mathrm{pc}}) \wedge Put_{\mathrm{pc}}(s) = k3 \\
\textbf{THEN } count(s) := count(s) + 1 \parallel Put_{\mathrm{pc}}(s) := k1 \\
\textbf{END}
\end{array}
$$

4.4 An Example Mapping to Java

The mapping of OCB to Java is mostly self-evident, and lends itself to automatic translation, since we have intentionally made OCB object-oriented. We now present an example mapping followed by a discussion of the translation rules.

The top-level OCB $MainClass$ construct maps to a Java class containing a main method which is used as the entry point for execution. More specifically the $MainClass$' non-atomic clause is mapped to the body of the *main* method. Each $ProcessClass$ in an OCB development maps to a Java class implementing the *java.lang.Runnable* interface; and the non-atomic (*na*) clause maps to the *run* method body. Each $MonitorClass$ maps to a Java class, but in this case the resulting classes should not implement the *Runnable* interface (since they do not behave as threads). Each of a monitor class' procedures map to a synchronized Java method, and these can then be called in $ProcessClass$ operation definitions. We now show the $Main$ class with its *main* method that arises from the translation of the $MainClass$,

```
public class Main{
 private static Put pu; private static Get ge;
 private static Shared sh;
 public static void main(String[] args){
    sh = new Shared();
    pu = new Put(sh);
    new Thread(pu).start();
    ge = new Get(sh);
    new Thread(ge).start();
 }
}
where vq = {v}
```

Attributes declared in the *MainClass* are mapped to static fields since they are referred to in the static *main* method. They could however be variables local to the *main* method but the choice appears to be arbitrary at this stage. They are restricted to private visibility to prevent their direct use externally; and this is true also of fields arising from *MonitorClass* and *ProcessClass* definitions. External access to monitor class fields is only through Java synchronized methods, which ensures their accesses are free from interference. We are able to see the synchronized methods of a monitor class in the following,

```
public class Shared{
  private int v; private int tally;
  public Shared(){ v = 0 ; tally = 0 ;}
  public synchronized int getVal(){
     try{while(!(v > 0))wait();}catch(InterruptedException e){...}
     tally = tally + 1 ; return v ;
  }
  public synchronized void setVal(int vl){ v = vl ;}
}
```

The class has a constructor where the fields are initialized; and two synchronized methods, one of which (*getVal*) contains conditional waiting. This arises from the mapping of an OCB *when* clause which is a blocking construct. Here the built-in Java *wait* method is used to block entry to the conditional critical region for as long as the condition for entry is not met. When the condition is met the conditional critical region is entered and processing proceeds. Some other thread will unblock the waiting thread using Java's built-in *notifyAll* method when an update is made to data held in the monitor.

We now look at the classes arising from the *ProcessClass* definitions,

```
public class Get implements java.lang.Runnable {
 private Shared shared; private int i; private int count;
 public Get(Shared shd){
    shared = shd; i=0; count=0;
 }
 public void run(){
    while(count<100){
       i=shared.getVal(); count=count+1;
    }
 }
}

public class Put implements java.lang.Runnable {
 private Shared shared; private int val; private int count;
 public Put(Shared shd){
    shared = shd; val=0; count=0;
 }
 public void run(){
    while(count<100){
       val=count % 5; shared.setVal(val); count=count+1;
    }
 }
}
```

The above classes declare *private* fields which are then initialized in the constructor method. Notice that the process classes implement the *java.lang.Runnable* interface, which requires us to provide an implementation for the *run* method.

4.5 Rules for Mapping OCB to Java

4.5.1 Overview

We now give an overview of some of the issues we consider when we map OCB to Java, before giving a more thorough description. In an OCB specification the actions of non-atomic clauses and procedure calls make use of the assignment operator ':=', when we maps to Java we substitute this operator for '='. When we do this we must accommodate the difference in the semantics of parallel composition of the action, and sequential composition of the Java statement. To do this we introduce temporary variables in Java that store the initial values upon method entry, and substitute these in certain places. When we implement the **when** clause, for conditional waiting, we must also consider the case where a waiting thread may be interrupted. In this situation a Java

InterruptedException is thrown, which must be caught by the waiting process. Another issue is that of implementing OCB guards, with respect to conditional operators, since their are several options one can choose from. In most cases operators of OCB guards are mapped to Java operators, for example the OCB equality operator '=' maps to '==' in Java and the negation $\neg(g)$ maps to $!(g)$, '/ =' maps to '! ='. We do however have a choice to make with the logical operators, '&' maps to Java's conditional-AND '&&'; we choose the conditional-AND option rather than the simple AND in order to make use of the optimisation, where the right-hand branch is not evaluated if it has already been found that the left-hand branch is false. In implementations the conditional-AND construct is often used to perform a well-definedness check using the left-hand branch, before evaluating the right branch. A typical example would be to check non-nullity of some target before making a call that is used in the evaluation of the right hand branch.

In the presentation of our translation from OCB to Java we apply the translation rules to the OCB model (a textual version of the OCB meta-model) to generate Java code. We begin at the top level with an OCB system, and apply a succession of transformation rules which act on the model elements, or collections of such elements. This effectively gives rise to textual substitutions that result in Java code. The OCB System S is comprised of a $MainClass$ named N and a $main$ operation mo; a set of process and monitor classes, $p_1 \,..\, p_n$ and $m_1 \,..\, m_n$ respectively; and a set of attribute declarations $sv_1 \,..\, sv_n$. Elements of the OCB model contained in angle brackets, <>, correspond to elements in the OCB meta-model (discussed further in chapter 5). Translation rules are defined over the meta-model elements, or sets of elements, and serves to clarify the link between the specification and implementation. We indicate that a rule $ruleName$ should be applied to an element $< el >$ using the notation $< el >^{ruleName}$. We define the following $*$ iterator that applies the rule to each element of a set or sequence. For a set with n elements we have $< el_1 \,..\, el_n >^{ruleName*} = < el_1 >^{ruleName} \,..\, < el_n >^{ruleName}$. We define $booleanLiteral$ as $\{TRUE, FALSE\}$, $integerLiteral$ which corresponds to the set of integers, and $stringLiteral$ for attribute and class names. The $sysDef$ rule applies appropriate rules to the constituent parts of S. In the following definitions we use italic font for meta-variables and translation functions, and standard font for concrete syntax.

$$
\begin{aligned}
&< S >^{sysDef} \\
&\triangleq \\
&< \textbf{MainClass}\ N\{\ sv_1 \,..\, sv_n \quad mo\ \} >^{mainDef} \\
&< p_1 \,..\, p_n >^{procSet*} \\
&< m_1 \,..\, m_n >^{monSet*}
\end{aligned}
$$

4.5.2 Mapping the MainClass to Java

The class that is the starting point for execution, i.e. with a *main* method, is generated from the attribute declarations $sv_1 \,..\, sv_n$ and *main* operation *mo* as follows,

$$< \textbf{MainClass}\ N\{\ sv_1 \,..\, sv_n \ \ mo\ \} >^{mainDef}$$
$$\triangleq$$
$$\textbf{public class}\ N\ \{$$
$$\quad < sv_1 \,..\, sv_n >^{svSet*} \quad < mo >^{moDef}$$
$$\}$$

where $mo \triangleq$ **Operation** $Main()\{\ na\ \}$ and *na* is a non-atomic clause. The translation of the main class' variable list gives rise to static fields; one for each attribute sv_i in the set of attributes $sv_1 \,..\, sv_n$, of type *sv*. Each OCB attribute declaration may have an optional initial value *init*. An attribute *sv* is defined using OCB notation as follows,

$$sv \triangleq type\ identifier\ [:=\ init]$$

where *type* is Integer, Boolean, or a class name (of type *stringLiteral*). The attribute *identifier* also is a literal string. An initial value must be supplied for Integer and Boolean types, although monitor and process class types have no initial value since they are to be instantiated in the non-atomic operation. In the case of array declarations no initial value is required since array initialisation is as per Java defaults, i.e. zero for **int**s, **false** for boolean, and **null** for class identifiers. We apply the rule *svSet* to each attribute sv_i in the set of attributes $sv_1 \,..\, sv_n$; we define *svSet* in the following way,

$$< sv >^{svSet}$$
$$\triangleq$$
$$\textbf{private static}\ \ < type >^{tDef}\ \ identifier\ \ < init >^{sviDef};$$

The translation from OCB types to Java types is shown in Table 4.2; defined types can be either simple types or array types. In our tables we use italic font for the variables of our translation, therefore in Table 4.2 *stringLiteral* is a place-holder for a string; in the translation $< type >^{tDef}$ the string is simply written, as text, in the Java source. However boolean is a string not a variable and that is written in the Java source as "boolean". Later in our work we will use *tDef* to translate the type of a returning procedure, in the case where there is no value to return we translate to **void**.

The translation of the initialisation depends on the type of the attribute involved; literals are unchanged, arrays use a constructor for initialisation, and any identifier is initialized to **null**. Integer and Boolean types take the value in the *init* clause. Initialisation is shown in Table 4.3

type	$< type >^{tDef}$
Integer	int
Boolean	boolean
stringLiteral	*stringLiteral*
	void
Integer[*integerLiteral*]	int[]
Boolean[*integerLiteral*]	boolean[]
stringLiteral[*integerLiteral*]	*stringLiteral*[]

TABLE 4.2: Rule tDef

type	*init*	$< init >^{sviDef}$
Integer	*integerLiteral*	= *integerLiteral*
Boolean	TRUE	= true
Boolean	FALSE	= false
stringLiteral		= null
Integer[*integerLiteral*]		= **new** int[*integerLiteral*]
Boolean[*integerLiteral*]		= **new** boolean[*integerLiteral*]
stringLiteral[*integerLiteral*]		= **new** *stringLiteral*[*integerLiteral*]

TABLE 4.3: Rule sviDef

The discussion of OCB *MainClass* attribute declarations is now complete and we turn our attention to the main operation. The main operation consists of a non-atomic clause *na*,

$$< \textbf{Operation}\ \text{Main}()\{< na >\} >^{moDef}$$
$$\triangleq$$
$$\textbf{public static void}\ \text{main(String[] args)}\{< na >^{naDef};\}$$

This leaves just the non-atomic clause to be expanded now, and we deal with this in the next section.

4.5.3 Mapping Non-Atomic Clauses to Java

The translation of the non-atomic clause in the main class using $naDef$ is shown in of Table 4.4. $naDef$ takes a non-atomic clause parameter and returns a *JavaStatement*. The translation for the main class *na* clause is the same as that of process class *na* clauses, so we can refer to this for processes also. The non-atomic clause *na* can be either a sequence, branching, looping or a labelled atomic clause *a*. Square brackets indicate features that are optional in OCB, during translation optional elements do not generate any Java code if they do not exist.

na	$< na >^{naDef}$
na_1 ; na_2	$< na_1 >^{naDef}$; $< na_2 >^{naDef}$
if(c) **then** a **andthen** na **endif** [**elseif**(c) **then** a **andthen** na **endelseif**] [**else** a **andthen** na **endelse**]	**if**($< c >^{cDef}$){ $< a >^{aDef}$; $< na >^{naDef}$ } [**else if**($< c >^{cDef}$){ $< a >^{aDef}$; $< na >^{naDef}$}] [**else**{ $< a >^{aDef}$; $< na >^{naDef}$ }]
while(c) **then** a **andthen** na **endwhile**	**while**($< c >^{cDef}$){ $< a >^{aDef}$; $< na >^{naDef}$ }
a	$< a >^{aDef}$;

TABLE 4.4: Rule naDef

The atomic clause a of Table 4.4 is a labelled atomic clause of which there are three main types; action, procedure call or create call. A procedure call, or a create call, may have a sequence of actual parameters ap which must match those of the target procedure definition in type and number. Procedure calls optionally have an assignment involving the return value, Table 4.5 shows the translation of the atomic clause types, where M, Q, m and x are *stringLiteral*s. M and Q represent a monitor class name and process class name respectively. m is an attribute name, pn is a procedure name, and x is an attribute name. $aDef$ takes an action as a parameter and returns a *JavaStatement*.

a	$< a >^{aDef}$
$m.pn(ap)$	$m.pn(ap)$
$x := m.pn(ap)$	$x = m.pn(ap)$
$x := M$.**create**(ap)	$x =$ **new** $M(ap)$
$x := Q$.**create**(ap)	$x =$ **new** $Q(ap)$; **new** Thread(x).start()
action	$(< action >^{acDef})^{iniSub}$

TABLE 4.5: Rule aDef

Actions are either a single assignment or a parallel composition of assignments and are translated using the $acDef$ rule shown in Table 4.6. The syntax of expression E, appearing on the RHS, is shown in appendix A. In expressions most OCB operators map directly to Java, the exceptions are the *mod* operator which maps to the % operator, and an exponent $x\hat{}y$ must be mapped to a Java statement (int) Math.pow(x,y), this returns a **double** value which we cast to an **int**.

Now there is a problem with mapping parallel composition to Java, consider the action where x is initially zero and then we perform the update $x := 1 \parallel y := x$. This cannot be simply mapped to $x = 1$; $y = x$ since in this case the semantics of parallel composition result in the post state $x = 1 \wedge y = 0$, but the incorrect Java mapping results in the

$action$	$< action >^{acDef}$
$action_1 \parallel action_2$	$< action_1 >^{acDef}; < action_2 >^{acDef}$
$x := E$	x=E

TABLE 4.6: Rule acDef

post state $x = 1 \wedge y = 1$. To rectify this we introduce local temporary variables which we declare and initialize on method entry. We then use these stored initial values in the execution of the remainder of the method body. In this way we are able to map OCB actions with parallel composition to sequential Java statements. To achieve this we introduce rule *iniSub* to apply to Java statements involving translations from actions, which is typed as,

$$iniSub \in JavaStatement \rightarrow JavaStatement$$

The translation of an OCB clause involving parallel composition takes place in two steps as follows. First we translate the OCB action into a partially complete Java statement which looks like the incorrect Java statement previously mentioned. We then collect all the variables v in statement s that appear on the LHS of an assignment and on the RHS of some other assignment; or on the LHS of an assignment and in a **return** statement. For each v; we declare and initialize a local variable $init\_v$ with the value of the variable before the update, and insert this at the beginning of the statement. The variables v are then substituted by $init\_v$ where they appear on the RHS of the other assignments or in the **return** statement. Note that if $x := x+1$ appears without x on the RHS of another assignment no initial value substitution is required. The OCB semantics, based on Event-B actions are equivalent to Java in this case, so $< x := x + 1 >^{acDef} \triangleq x = x + 1$. Returning to the example, using initial value substitution we introduce the rule *iniSub* and apply it after applying $acDef$,

$$(< x := 1 \parallel y := x >^{acDef})^{iniSub} \triangleq init\_x = x \ ; \ x = 1 \ ; \ y = init\_x$$

We now describe the rules for translating parallel composition to sequential Java. We apply a rule called *iniSub* to a partially translated Java statement s. This rule inserts the initial variables and replaces the appropriate variables on the RHS of assignments. Assume we have a number of variables v in a statement s to rewrite. We introduce local variables $lv_1 .. lv_n$, one for each variable v that we need to rewrite. We declare and initialize each lv_i of $lv_1 .. lv_n$ to the initial value of v_i. Here n is the number of variables v appearing on the LHS of the assignment statement and also on the RHS of some other assignment in s, or on the LHS of an assignment and also the **return** statement. Each local variable lv_i then replaces v_i where it appears on the RHS of an assignment. More formally $lv_1 .. lv_n$ maps to a sequence of semi-colon separated declaration/initialisation

statements,

$$< lv_1 \,..\, lv_n >^{lvSet} \triangleq t_1\ lv_1\ =\ v_1\ ;\ \ldots\ ;\ t_n\ lv_n\ =\ v_n \tag{4.2}$$

where: for each variable v_i, t_i refers to the Java type of v_i.

We use $< lv_1 \,..\, lv_n >^{lvSet}$ to insert the initialisations at the beginning of statement s as follows,

$$\begin{array}{l} < s >^{iniSub} \triangleq \\ < lv_1 \,..\, lv_n >^{lvSet}\ ;\ s[v_1 \ldots v_n \backslash lv_1 \ldots lv_n]^{RHS} \end{array} \tag{4.3}$$

where: $v_1 \ldots v_n$ are the variables appearing on the LHS of an assignment and the RHS of another assignment. The substitution $[\ldots]^{RHS}$ is substitution restricted to the RHS of assignment statements.

Conditional statements are used in OCB's branching, looping and conditional waiting clauses. The translation of each OCB *condition* to a *JavaStatement* is presented in Table 4.7. Conditional statements may involve expressions E and guards *Guard*, as shown in appendix A. The mapping here is relatively straightforward.

c	$< c >^{cDef}$
$E\ =\ E$	$E\ ==\ E$
$E \neq E$	$E\ !=\ E$
$E < E$	$E < E$
$E > E$	$E > E$
$E \leq E$	$E\ <=\ E$
$E \geq E$	$E\ >=\ E$
$Guard \wedge Guard$	$Guard\ \&\&\ Guard$
$Guard \vee Guard$	$Guard\ \|\|\ Guard$
$\neg Guard$	$!(Guard)$

TABLE 4.7: Rule cDef

4.5.4 Mapping a ProcessClass to Java

We apply *procSet** to generate a Java class for each OCB process class declaration p_i, named P_{name}, in the set $p_1 \,..\, p_n$ of process classes. Using OCB notation a process class p is defined as follows,

$$p\ \triangleq\ \textbf{ProcessClass}\ P_{name}\{\ v_1 \,..\, v_n\ c\ r\ \}$$

and where $v_1 \,..\, v_n$ is a set of n attribute declarations, c is the *create* procedure, and r is the *run* operation. Then the mapping to Java follows where *procSet* takes a process

class definition as a parameter and returns a Java class definition,

$$
\begin{array}{l}
< p >^{procSet} \\
\triangleq \\
\textbf{public class } P_{name} \textbf{ implements } \text{java.lang.Runnable}\{ \\
\quad < v_1 \,..\, v_n >^{vSet*} \quad < c >^{crDef} \quad < r >^{prDef} \\
\}
\end{array}
$$

We first look at the mapping of the attribute declarations which give rise to variable declarations in Java. These are similar to, but slightly simpler than, the equivalent *MainClass* attribute declarations since we have no OCB initialisation to consider. All explicit OCB initialisation is done in the create method, however certain Java fields need to be initialized as detailed in Table 4.8 below. Assume each attribute declaration v_i, of type v, in the set of variables $v_1 \,..\, v_n$ consists of a *type* and name *identifier* defined using OCB notation as follows, where *type identifier* is sugar for $identifier \in type$,

$$v \triangleq type\ identifier$$

The *type* can be Integer, Boolean, or a class name (of type $CName$). In the case of array declarations, and class names we need to perform initialisation in the Java even though none is explicitly specified in OCB. We apply the following rule to map each attribute v_i in the set of attributes $v_1 \,..\, v_n$,

$$
\begin{array}{l}
< v >^{vSet} \\
\triangleq \\
\textbf{private} \quad < type >^{tDef} \quad identifier \quad < init >^{viDef} \quad ;
\end{array}
\tag{4.4}
$$

The translation from OCB types to Java types using $tDef$ is shown in Table 4.2, and we show the initialisation part in Table 4.8 where types can be either simple types or array types. In the case of array initialisation, a new array is constructed using an array constructor which is invoked using the **new** keyword.

type	$< init >^{viDef}$
Integer	
Boolean	
$CName$	= null
Integer[$integerLiteral$]	= **new** int[$integerLiteral$]
Boolean[$integerLiteral$]	= **new** boolean[$integerLiteral$]
$CName$[$integerLiteral$]	= **new** $CName$[$integerLiteral$]

TABLE 4.8: Rule viDef

We now consider the create procedure c of a process class named P_{name}, the create

method consists of an action a containing assignments used for attribute initialisation, and a sequence of formal parameters $fp_1 .. fp_n$. Using OCB notation the create clause c is defined as follows,

$$c \triangleq \textbf{Procedure}\ \ \textbf{create}(< fp_1 .. fp_n >)\{\ \ < action >\ \ \}$$

Translation $crDef$ takes a create procedure as a parameter and returns a Java constructor method, applying $crDef$ to the create clause c gives rise to the following,

$$< c >^{crDef} \ \triangleq\ \textbf{public}\ P_{name}(< fp_1 .. fp_n >^{fpSeq*})\{\ \ < action >^{acDef}\ \ ;\ \} \qquad (4.5)$$

Here the translation of *action* using $acDef$ is shown in Table 4.6. $acDef$ takes an *action* parameter and returns a JavaStatement. The type information associated with each formal parameter needs to be mapped to a Java type. We apply $fpSeq$ to each of the n formal parameters fp_i, of type fp, in $fp_1 .. fp_n$. In OCB each formal parameter fp defined as a *type* and an $identifier$,

$$fp\ \triangleq\ type\ identifier$$

applying $fpSeq$ to each fp we obtain,

$$< fp >^{fpSeq}\ \triangleq\ < type >^{tDef}\ \ identifier \qquad (4.6)$$

The identifier is written to Java source with its type derived from rule $tDef$ of Table 4.2.

The concluding part of the discussion about process classes involves mapping of the *run* operation r. We define the *run* operation r using OCB notation as follows,

$$r\ \triangleq\ \textbf{Operation}\ run()\{< na >\}$$

$prDef$ takes the *run* operation as a parameter and returns a Java method, so applying $prDef$ to r we have,

$$< r >^{prDef}\ \triangleq\ \textbf{public}\ \ \textbf{void}\ \text{run}\{< na >^{naDef};\}$$

and $naDef$ was described in Table 4.4.

4.5.5 Mapping a MonitorClass to Java

To translate each OCB monitor class m_i, of type m, in $m_1 .. m_n$ with name M_{name}, we apply the rule $monSet$ to each m_i. This maps each m_i to a Java class. Using OCB

notation the monitor class definition m is defined as follows,

$$m \triangleq \textbf{MonitorClass}\ M_{name}\ \{\ <v_1\ ..\ v_n>\quad <c>\quad <pr_1\ ..\ pr_n>\ \}$$

where $v_1\ ..\ v_n$ is a set of attribute declarations, c is the *create* procedure and $pr_1\ ..\ pr_n$ is the set of procedures. We have the translation function *monSet* which takes a monitor class definition as a parameter and returns a Java class definition, applying *monSet* to m we have,

$$\begin{aligned}
&<m>^{monSet}\\
&\triangleq\\
&\textbf{public class}\ M_{name}\ \{\\
&\quad <v_1\ ..\ v_n>^{vSet*}\quad <c>^{crDef}\quad <pr_1\ ..\ pr_n>^{mprSet*}\\
&\}
\end{aligned}$$

We apply *vSet* to each of the attributes in $v_1\ ..\ v_n$ as shown in equation 4.4, and *crDef* to the create clause as shown in equation 4.5. It just remains then to discuss the transformation *mprSet* which is applied to each procedure pr_i of type pr in the set of procedures $pr_1\ ..\ pr_n$. We define the procedure pr as follows,

$$pr \triangleq \textbf{Procedure}\ Pr_{name}(<fp_1\ ..\ fp_n>)\{\ <mpb>\ \}:\ type$$

A procedure pr has a name Pr_{name}, a sequence of formal parameters $fp_1\ ..\ fp_n$, and a monitor procedure body *mpb*. The monitor procedure may contain a conditional wait construct, and may also return a value. *mprSet* takes a monitor procedure as a parameter and returns a Java method. So for each procedure pr_i, of type pr, in $pr_1\ ..\ pr_n$ we apply *mprSet* as follows,

$$\begin{aligned}
&<pr>^{mprSet}\quad \triangleq\\
&\textbf{public synchronized}\ <type>^{tDef}\ \ Pr_{name}(<fp_1\ ..\ fp_n>^{fpSeq*})\{\\
&\quad (<mpb>^{mpbDef})^{iniSub}\\
&\}
\end{aligned}$$

Once again we can refer to previously defined rules for $tDef$ in Table 4.2, $fpSeq*$ in equation 4.6, and $iniSub$ in equation 4.3; it then only remains to define the rule $mpbDef$ where we need to accommodate conditional waiting, and assignment to the reserved attribute **return**. The monitor procedure body *mpb* is defined using OCB notation as follows, the square brackets indicate the optional **when** clause,

$$mpb \triangleq [\ \textbf{when}(c)\{\ ]\ <action>\ [\ \}\ ]$$

We first look at the translation for an *mpb* clause that does not a **when** clause, $mpbDef$ takes a conditional waiting clause as a parameter and returns a Java statement.

$$
\begin{array}{l}
< mpb >^{mpbDef} \\
\triangleq \\
< \\
\quad < action >^{actDef}; \\
\quad \text{notifyAll();} \\
\quad < action >^{retDef}; \\
>^{iniSub}
\end{array}
$$

The body of the action may take the forms shown in Tables 4.9 and 4.10, note the return statement is removed from the Java in $actDef$ since $retDef$ is used to extract this, and insert it as the last Java statement. The **return** assignment, if one exists, can appear anywhere in the action since it is not composed sequentially. The final step is to gather the initial values and make appropriate substitutions on the RHS of assignments, where applicable, by applying $iniSub$.

action	$< action >^{actDef}$
$< action > \parallel < action >$	$< action >^{actDef}; < action >^{actDef}$
$x := E$	x=E
return $:= E$	

TABLE 4.9: Rule actDef

action	$< action >^{retDef}$
return $:= E$	**return** E

TABLE 4.10: Rule retDef

Now we apply $mpbDef$ to a monitor procedure body mpb that does have a **when** clause,

$$
\begin{array}{l}
< mpb >^{mpbDef} \\
\triangleq \\
< \\
\quad \textbf{try}\{ \\
\quad\quad \textbf{while}(! < c >^{cDef})\{ \\
\quad\quad\quad \text{wait}(); \;\; < lv_1 \,..\, lv_n >^{lvSet}; \\
\quad\quad \} \\
\quad \} \\
\quad \textbf{catch}(\text{InterruptedException e})\{ \\
\quad\quad \text{e.printStackTrace}(); \\
\quad \} \\
\quad < action >^{actDef}; \\
\quad \text{notifyAll}(); \\
\quad < action >^{retDef}; \\
>^{iniSub}
\end{array}
$$

In the translation of the **when** clause to Java we use the previously defined rules, $lvSet$ of equation 4.2, and $cDef$ of Table 4.7. $lvSet$ is applied after the return from the *wait* clause. We reset the initial values when a thread resumes in case any of the variables have changed by some other process while waiting. The final step is to gather the initial field values and make appropriate substitutions on the RHS of assignments where applicable by applying $iniSub$ to the remainder of the statement.

4.6 Review of the Chapter

In this chapter we finalized our main contribution. We showed how, using an intermediate specification notation, we could link an Event-B development with a contemporary object-oriented, concurrent programming language. The main focus of the contribution is the ability to specify concurrently executing processes in the target implementation. Although we link to an object-oriented programming language, and OCB has an object-oriented look and feel, we have not explored the use of other object-oriented features such as inheritance. The Event-B language does not contain object-oriented constructs or low-level programming constructs; nor does it handle interleaving operations, so using OCB we have been able to link these notions with Event-B and produce a mapping to Java code. We introduced the following features to OCB to facilitate the link between Event-B and concurrent, object-oriented implementations; $ProcessClass$ - as templates

for process objects, *MonitorClass* - as templates for monitor objects, constructor procedures - to instantiate the objects, and OCB- arrays.

OCB provides constructs for defining process classes that can be instantiated and implemented as Java threads, and monitor classes that use the Java synchronization mechanism to ensure mutually exclusive access to shared data. The limitations of the Java Language specification JLS 2 have been taken into consideration and the approach has been tailored to accommodate this particular version. A feature of the OCB notation is that a useful abstraction is made which alleviates the developer from the burden of reasoning about locking and implementing conditional waiting, and simplifies the reasoning process due to the nature of the labelled atomic clauses. One of the key features is that of OCB's ability to specify interleaving operations; using non-atomic clauses comprised of labelled atomic constructs. The Event-B language does not provide constructs to enable specification of interleaving in the same way. The non-atomic clauses facilitate sequential composition, looping, and branching behaviour. These are also features which are absent from the Event-B language. In the presentation of these non-atomic constructs we initially use the Guarded Command Language to describe the semantics. However, the guarded command language is not used for the specification notation itself, instead we introduce syntactic sugar which is more affine to modern object-oriented notations. In this way we introduce the specification style of constructs such as **while** and **if**. We then proceed to use this syntax to define the mapping to Java code, and handle the issues of parallel semantics for sequential Java statements, using initial value substitution. An alternative approach here would be to introduce a sequential operator for use in Event-B actions; since one does not exist at present. This would facilitate a more straightforward mapping from Event-B to Java, but would require a change to the Event-B language to incorporate a sequence operator.

In this chapter we introduce the OCB **when** construct, of a procedure call. This construct is a simple representation of a guarded event in Event-B, and maps to a Java style conditional waiting loop in the implementation. The negated **when**-clause condition is used as the condition for entry to the **while** loop invoking the *wait* method. Notification statements are added to all procedures that update a monitor, in this way waiting processes are informed of potential enabling conditions. The restrictions noted in earlier chapters are eased somewhat in the extension presented in Chapter 8, in which we introduce transactional constructs, which allow access to multiple, shared objects. Formally verifying that the Java code is correctly synchronized and correctly corresponds to the formal model is the subject of future work.

Chapter 5

Tool Support for OCB

In this chapter we describe the tool support provided for the approach. We provide an overview of the Eclipse Platform, and some Eclipse projects that we make use of. We describe the OCB meta-model which we use to create OCB models in Eclipse; the RODIN platform and Event-B meta-model that is the target for formal analysis; and the Java Development Toolkit, which has a Java meta-model that we use to create a Java implementation.

In previous chapters we presented a textual version of Event-B. Our translators however integrate with the RODIN tool, and to-date the tool has no text based editor. The RODIN tool is a GUI based modelling environment [9], for Event-B, based on the Eclipse platform [148]. It is designed to be an open, extensible platform for rigorous development of complex systems. We extend the functionality of the platform with plug-ins that facilitate specification, and translation, of OCB models. We translate to the target Java implementation; where the Java files reside in the Eclipse workspace in a Java Project. We also translate to an Event-B project file, where the Event-B model elements reside in a RODIN database (this is an Eclipse Project with some additional information attached). The formal model will be amenable to analysis with the Event-B tools and the Java project will run as an application in Eclipse.

5.1 An Overview of Eclipse

Eclipse is an open source project that provides an extensible tool platform that functions as a customizable Integrated Development Environment (IDE). A key feature of the Eclipse platform is the support for extensibility through the use of plug-ins, and this was one of the main justifications for the use of Eclipse as the basis for the RODIN tool. We aim to develop plug-in extensions for an OCB meta-model, this will allow creation and editing of OCB models in the Eclipse environment. We also develop a plug-in to

facilitate translation of an OCB model into an Event-B model which resides in a RODIN database, and to Java source code. In order to provide a plug-in we make use of the Eclipse Software Development Kit (SDK).

5.1.1 The Eclipse Software Development Kit

The Eclipse platform provides the tools to discover and utilize plug-ins, and also provides mechanisms for viewing and manipulating resources. We will develop a plug-in using the facilities provided by the Plug-in Development Environment (PDE) which is part of the SDK. By making use of the PDE we significantly reduce the amount of coding effort required, since the PDE contains wizards for assisting with creation of plug-ins. An example of another productivity enhancement is the use of an extension mechanism, called extension points, which allow the re-use of previously written code. We use extension points to add pop-up menus and menu-bar items to the user interface. When implementing a pop-up menu extension, for instance, most of the code is created by the wizard and it remains for us to specify the desired behaviour in the *run* method of a class that implements *IObjectActionDelegate*. Appendix E.1 shows the *run* implementation for our translator plug-in.

At run time, a user can build a model using Eclipse tree editor; and when they wish to translate to Event-B and/or Java they can select the $Translate$ pop-up menu option which is enabled when right-clicking over an OCB model. At this point some diagnostics are invoked, to check the integrity of the model. The information about which elements are allowed, and which elements must be present, are specified in the meta-model itself. So for example we stipulate that a system must have exactly one $MainClass$ element, and the $MainClass$ must have a name and a non-atomic clause. Following successful diagnostic checks the Event-B translator and then Java translator are invoked.

5.1.2 The Eclipse Modelling Framework

In our work we have defined the OCB syntax, and we wish to carry out some modelling activity, creating OCB models in accordance with the syntax we have described. The Eclipse Modelling Framework (EMF) [150, 27] is equipped to provide exactly this facility. In essence the OCB syntax that we have described will be embodied in a meta-model, and we can use the EMF to create an editing environment that will enable a user to construct OCB models using the meta-model.

Before we proceed we will clarify the terms that we are using in the discussion of EMF, we frequently use the terms model and meta-model. A model is a structured description of some artefact, and the meta-model describes the elements that will be used to build that model, (a meta-model is also a model). Therefore a meta-model describes the

structure of a whole range of models. The EMF and its associated tools allows us to build an OCB meta-model; and the OCB meta-model is then used to create an OCB model. We then use this model to translate to Event-B and Java.

In Eclipse, using the EMF meta-model creation wizard, there are a number of ways of generating a meta-model; the source for generation can be an existing UML model, annotated Java source code, or an XMI (XML Meta-data Interchange) document. We chose to use annotated Java due to its familiarity and, with regard to the UML approach, to reduce the reliance on other technologies. The meta-model is generated with two additional plug-ins. The first is the *edit* plug-in which contains item providers for the meta-model elements, this is used by EMF to create model elements when constructing an OCB model using the GUI. The second is the *editor* plug-in, which is used to contribute to the User Interface (UI). The *editor* plug-in contains an EMF model creation wizard for creating new modelling projects, and it also contains classes that contribute to the toolbar and pop-up menu.

5.2 The OCB Meta-model

We now discuss the creation of the OCB meta-model, as discussed previously, we chose to use annotated Java as the basis for meta-model construction. When using annotated Java as the basis for meta-model creation we define Java interfaces for the meta-model elements we wish to create, and provide getter methods for the child elements. The EMF meta-model creation wizard then uses this information to create an *ecore* and EMF model. During the development of the OCB meta-model we found that the syntax mapped easily to the annotated Java. Figure 5.1 shows the annotated Java specification the we use to construct the *ProcessClass* element of the OCB meta-model. Figure 5.2 shows the relationship between the OCB syntax and elements in the annotated Java. Some explanation of the annotation follows: The *@model* annotation is used to identify the meta-model elements of the *ProcessClass* Interface (including the interface itself) that should be included in the meta-model; parts of the interface without annotation will not be processed. The *@model* annotation associated with the *ProcessClass* Interface will give rise to an *ProcessClass* meta-model element that can be instantiated within the Eclipse environment as a model element. The $required = true$ annotation is an instruction to the model validator to check that an element of this type is present in the model, and raise an error if it is not present. The *containment* annotation is an instruction to the generator of the editor. $containment = true$ gives rise to an element which can be added to the model tree in the tree editor view; if it is false, or not specified, the element appears in the properties view editor and not in the model tree.

The *ProcessClass* defined in Figure 5.1, gives rise to the implementation classes shown in the class diagram in Figure 5.3. We see that the implementation classes have been

```
/**
 * @model
 */
public interface ProcessClass {
 /**
  * @model required="true"
  */
 String getName();

 /**
  * @model type="Variable" containment="true"
  */
 EList<Variable> getVariable();

 /**
  * @model required = "true" containment="true"
  */
 NonAtomicClause getRunOperation();

 /**
  * @model containment="true"
  */
 CreateProcedure getCreateProcedure();
}
```

FIGURE 5.1: Annotated Java for the ProcessClass Meta-model Element

OCB Syntax	Annotated Java
ProcessClass	Interface ProcessClass
CName	String getName()
Var$^+$	EList<Variable> getVariable()
Constructor	CreateProcedure: getCreateProcedure()
NonAtomic	NonAtomicClause: getRunOperation()

FIGURE 5.2: Relationship between OCB Syntax and Meta-model

populated with various attributes and operations by the meta-model generator. These are used by the tree editor when creating and managing the model elements.

We present an illustration of the use of the OCB meta-model, an OCB model, in Figure 5.4. The screen-shot shows the tree editor view of a model described later, in chapter 6.1. In the tree view the details of the specification are distributed throughout the nodes of the tree - many of which are likely to be hidden at any one time in anything other than the most simple development. In addition part of the specification appears in a totally separate view to the tree editor - the properties view. So, as we see, the tree editor view may not be so easy to understand, especially when trying to convey information about the development to others. So in addition to the tree-editor view we

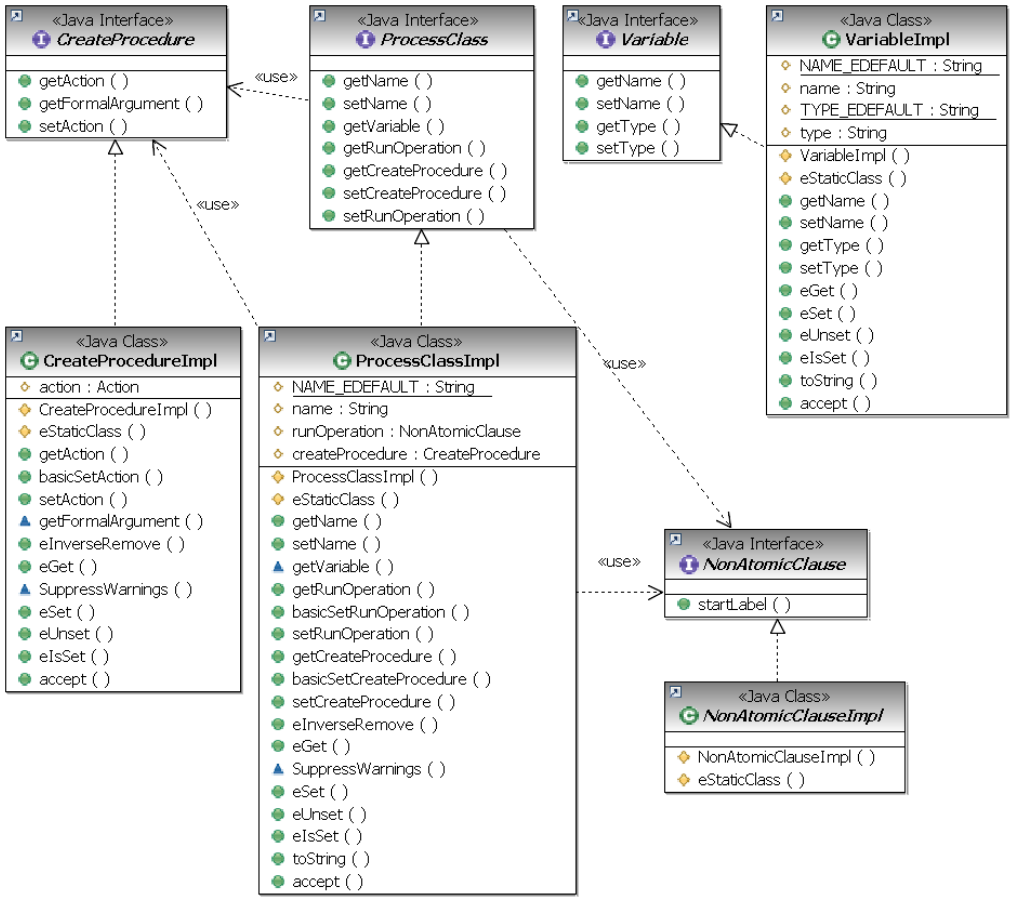


FIGURE 5.3: Meta-model - Class Diagram of the ProcessClass

have provided a utility to produce a text based view of the development, also shown in the same figure. The textual view makes use of the syntactic sugar described in section 4.2 and provide a pop-up menu item to translate the tree-based model to a text file. In future work we would like to develop a text-based editor for OCB with syntax highlighting and context sensitive code assists using a framework such as the Textual Editing Framework (TEF) [106].

When a user wishes to create a new OCB model a new, empty model is created using the model creation wizard built by the framework. Once created new model elements can be added to the model using the tree editor view. The underlying framework ensures that model elements can only be added to the appropriate tree node. To complete the description of the OCB development, information is entered in the properties editor view, which is visible in the properties tab at the bottom of Figure 5.4. When the model is ready for translation it can be validated using the *validate* menu option by right-clicking on the *MainClass*, although the translator enforces validation of the model programatically before any actual translation takes place.

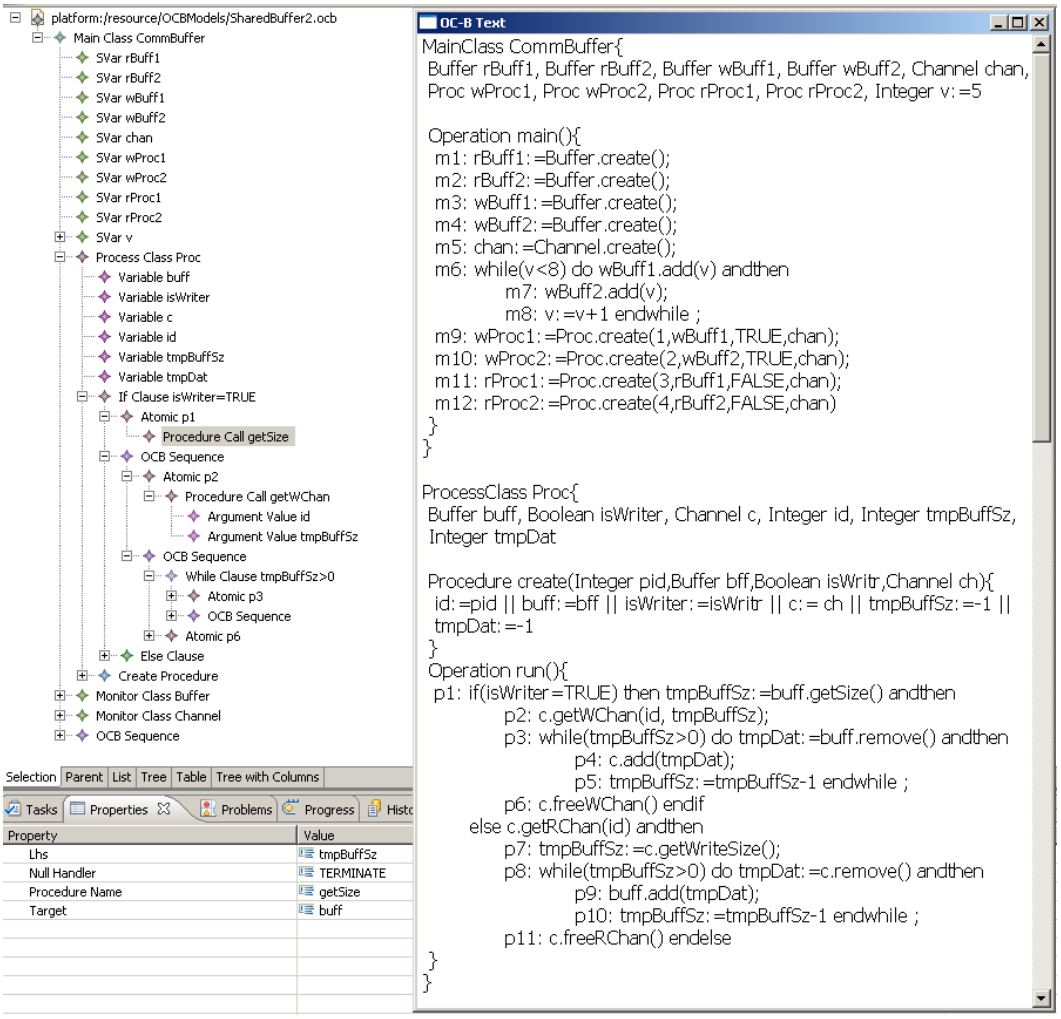


FIGURE 5.4: Comparison between OCBText and Tree Editor

5.3 Implementing the OCB to Event-B Translator

When translating from OCB to Event-B we use a number of the published APIs from the RODIN tools project, which we use to create and populate a RODIN database. The model that results from running the Event-B translator on an OCB model, resides in an Event-B project (the so-called database) which is created programatically by the translator. The RODIN tool is comprised of a number of plug-ins, and we frequently use the API defined in the Event-B core plug-in. The core plug-in defines elements for use in an Event-B development, that is, the elements that are used to build Event-B models such as events, guards and proof obligations.

In order to automatically translate an OCB model to an Event-B model it is necessary to programatically create a RODIN project, populate an Event-B model with its representation of the OCB model, and make the result persist. A reference to the RODIN database manager is obtained, and this is used to create a new RODIN project which appears in the Eclipse GUI as a folder. An attempt is made to retrieve the new RODIN

project's Eclipse platform project (if it does not already exist it is created) and this is used to open the project for reading and updating. Once open we check that a RODIN project nature is associated with it. This is used by the Eclipse platform to recognise a RODIN project, and the platform then invokes the RODIN builders when changes are detected. The open project is then ready to be populated with RODIN elements such as machines, contexts, variables and events etc. Following translation the files in the RODIN project are saved using RODIN API calls.

The translation process is initiated by right clicking on the OCB model in the Eclipse tree editor and selecting the *OCB Utilities/Translate* option from the pop-up menu. When the *Translate* menu option is selected a listener causes the *TranslatorAction*'s *run* method to execute see Appendix E.1 for details. A diagnostic test is performed to ensure required elements are present, and if successful we create two translation managers; the $EventBManager$ that manages the OCB to Event-B translation; and the $JavaManager$ that manages the translation from OCB to Java.

The Event-B translation manager sets up the Event-B project files and a *translator* instance is created to traverse the model tree, and create the appropriate Event-B elements. The translation steps are summarized as follows,

1. Create the project, machine, and context files.
2. Prepare the project environment, add a machine file and context file.
3. Create the translator instance.
4. Invoke the translator's translate method to add elements to the Event-B components.
5. Save the project and components.

During translation from OCB to Event-B we add a machine and context, and traverse the OCB model, adding sets and related axioms, variables and their typing invariants, and events as appropriate using RODIN API methods. The formal presentation of the translation rules of Chapters 3 and 4 elides the exact details of these additions in-order to clarify the fundamentals of the approach. In future work we would wish to fully define these translation details and explicitly link each translation rule to the code that implements the rule, using code comments, for traceability.

As the translator traverses through the OCB model, OCB model elements are identified that contribute to the Event-B model; after processing the monitor classes the translator then processes the process classes. (Monitor classes are used by process classes so its easier to process these first). The processing of non-atomic clauses depends on the type of clause involved, so we find the instance type and process the specific type accordingly.

For instance, if the clause is an OCB sequence we recursively process the left and right branch non-atomic clauses. See Appendix E.2 for the Java method for processing the sequence clause.

5.4 Implementing the OCB to Java Translator

In the translation to Java we use the Eclipse Java Development Toolkit (JDT) API [149] to create and populate a Java project with Java source code. The JDT contains a meta-model of Java elements such as packages, classes, fields, and so on, which can be instantiated to programatically build Java developments. Using our translator we traverse the underlying OCB model and extract the information required to create the appropriate Java elements using the JDT API. The first step is to create a Java project within the Eclipse environment. The project is then populated with a number of compilation units created using the JDT *org.eclipse.jdt.core* class factory methods. The Java elements created by these factory methods are the programming elements that we wish to add, such as class declarations, field declarations, method declarations and method calls, imports statements, and looping and branching constructs.

The translation to a Java project is initiated by calling the *JavaManager*'s *translate* method. As previously mentioned we traverse the OCB model building and populating a Java project, and instantiate the meta-model elements exposed by the JDT API. The mapping from our object-oriented OCB constructs to their counterparts in Java is relatively straightforward, accommodating parallel semantics is one of the few complexities. Creation of a Java program that is executable in the Eclipse environment involves a number of tasks in addition to the translation task. The following steps are carried out programatically,

1. Create an Eclipse project and apply the JDT nature.
2. Use the Eclipse project to create a Java Project.
3. Add the Java Runtime's location to the Java project class path.
4. Create a Package to contain the source code.
5. Prepare the environment.
6. Traverse the OCB model, adding elements to the compilation unit. Each *ProcessClass*, *MonitorClass* and the *MainClass* gives rise to a Java File containing a Java class. The Java classes are populated programatically with Java Fields, methods and supporting elements.

When programatically adding elements to the Java project using the JDT API behind the scenes the AST Parser is syntax checking on the fly as we build a model. For this reason it is possible to get Java syntax errors mid-way through the programmatic construction of a development. The current tool sometimes terminates the translation with a cryptic error message. In future work it will be necessary to improve the handling of these errors, although we should be able to avoid the majority (if not all) of them through improved static checking of the OCB model prior to translation.

The *JavaCore* API provides various utilities for creating Java elements from which we obtain a *JavaProject* which appears in the Eclipse GUI as a folder. A Java package is then added to the *JavaProject* which is where the source files that we create will reside. A translator instance is created by the translation manager to populate the model with Java elements. The translator instance creates files and classes (from the org.eclipse.jdt.core package) that correspond to *ProcessClass*, *MonitorClass*, and the *MainClass*.

The compilation unit is the in-memory representation of a Java *.java* file, and has a method to create a class (known as a *type* in the JDT) by invoking the *createType* method. Typically when creating elements using the JDT API a parent is responsible for creating its child elements, to populate a class with fields we use a similar approach using *IType.createField*, and so on for other JDT meta-model elements. The process of building JDT models is slightly different to that of building Event-B models. When creating Event-B elements, factory methods provide a handle to the new object, a *create* method is then used to instantiate the element, then element properties are set explicitly using setter methods. However, when creating JDT elements we provide the API *create* method with a string parameter containing the Java source code. The new Java element is then added to the compilation unit, and source code is parsed using the AST parser. When the translation is complete the Java files are compiled by the JDT builder into ".class" files. These are suitable for execution in the Eclipse environment. The Java program can then be run in the Eclipse environment by identifying the class which contains the *main* method, derived from the MainClass *main* operation, and then selecting *Run As/Eclipse application*.

5.5 Review of the Chapter

In this chapter we have shown how tool support is provided for the OCB approach. We begin with an overview of the Eclipse platform, and the Eclipse Modelling Framework, upon which our tool is based. We then discuss the construction and use of the OCB meta-model, which provides the abstract syntax for our implementation level models. The OCB model is traversed by the OCB Event-B translator, it makes calls into the RODIN API to create an Event-B model that is compatible with the RODIN toolset. A

similar translator traverses the OCB model and uses the JDT API to create and populate a Java project. This can then be run in a JVM, or in the Eclipse environment. During initial investigations we found construction of OCB models relatively straightforward; and translations of the OCB models produced Java code that was executable. The resulting Event-B models were amenable to syntax checking and proof.

The current version of tools does not provide seamless integration when refining an existing development. The Event-B model arising from a translation is manually copied to an existing project after which work on the refinement activity continues. In future work we would aim to provide a more integrated approach to providing implementation refinements of an existing development. There is no plan to achieve tighter integration using round-trip engineering, due to the difficulty of mapping from an Event-B model to OCB. For instance when a guard is added to an Event-B model it would unclear where this would be added in the OCB; a single event may have guards derived from several locations in the OCB. A closer integration with UML-B could be beneficial, due to the potential link between UML-B classes, and $ProcessClass$ and $MonitorClass$ constructs, in class diagrams for instance. There are links between an abstract development and implementation refinement which can be brought into the foreground (of the developer's attention), perhaps using a GUI wizard to link the two. One example of where this would be of use is when defining the **REFINES** link between events of the abstract development and those of the OCB model; another would be the definition of witnesses using a **WITH** clause of some of the refined events of the OCB model. Witnesses link the parameters of the abstract development with the implementation refinement, and can assist with discharging proof obligations.

In our current tool we have made use of the default Eclipse tree based editor which provides the most basic editing functionality, such as addition and removal of nodes representing model elements, and a properties editor which allows editing of the properties associated with model elements. To assist with understanding a model we provide a textual OCB viewer, and in the future it would be useful to produce an OCB text editor, such as one based on the Textual Editing Framework (TEF) [106], to compliment the tools. A further enhancement would be to use the Eclipse Graphical Editor Framework (GEF) [14] to construct a GUI which will allow the user to construct models in an environment that uses entities similar to class diagrams.

We found that so far, in the development of the prototype tooling, that the OCB approach was not adversely affected by the use of EMF as a method of constructing a meta-model; in fact the Eclipse tools appear to be a good facilitator for the approach, with most of the OCB syntax embodied in the OCB meta-model. The translators, similarly, are facilitators of the approach and there was no significant impact on the approach due to the nature of the translator implementation, although integration with abstract developments was not attempted. It is certainly likely that with closer integration between abstract developments and OCB modelling then the use of a more

advanced features will feed back into the approach. This may be through the use of patterns, or other productivity enhancements to improve the link between the abstract development and OCB model.

Chapter 6

Case Study 1

In this chapter we present a case study of a channel buffering system. We use the case study to introduce a systematic approach to linking an Event-B model to an OCB specification. A diagrammatic view of the system is presented which follows the proposal in [32] of Event Refinement Diagrams. Event Refinement Diagrams are in turn based on Jackson Structure Diagrams [86]. These act as an aid to visualising the relationship between the events of the abstract development and the OCB specification. We partition the system into processes that perform tasks, and shared data structures. In our case study we specify a shared channel which is able to hold a block of data. Processes read data into a local buffer from the shared channel, or write data to the shared channel from a local buffer. We begin with an abstract model which models transfer of one block of data at a time, before refining the model based on the transfer of the packets that make up a block. Then we link the refined model (of the abstract development) with an OCB specification, which facilitates translation to an Event-B implementation model and Java code.

6.1 Development of a Concurrent Read/Write Channel

The channel that we specify will have at most one reader reading, and at most one writer writing at any one time; however a number of processes may be waiting to read from, or write to, the channel. An important feature of our system is that at the highest level of abstraction data is transferred as a block in a single atomic step. A write event constitutes moving a block from a writing process to a channel buffer; and a read event constitutes moving a block from a channel buffer to a reader. The atomicity of the read and write activity is altered in the refinement - we introduce blocks that are made up of packets, and each packet is written to the channel individually. This allows the reader to begin reading as soon as there is data in the channel - without the writer having to complete the data transfer. We also add an additional constraint that the

reader finishes reading a writer's data before another writer can begin writing. In effect, we require a block of data to be moved from one writer, to one reader, via a shared channel buffer. Figure 6.1 shows a configuration with one reader, one writer and a number of processes waiting. In order to describe the activities that occur we make

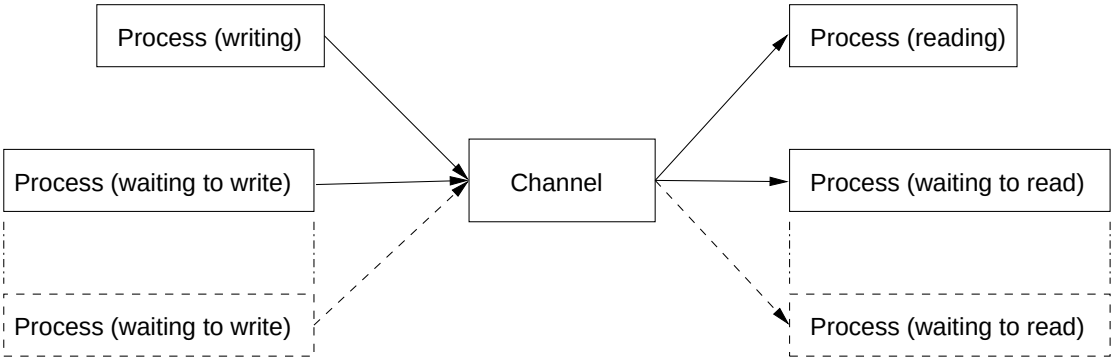
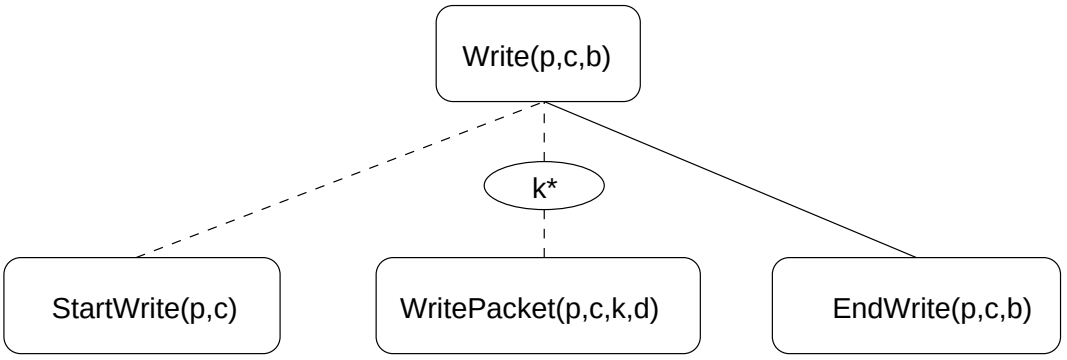


FIGURE 6.1: The Processes Sharing a Channel

use of a graphical representation of our system, based on Jackson Structure Diagrams. The diagrams are an informal representation of the relationship between abstract events and events of refinements; and are used as an aid to understanding the correspondence, in particular where an event refinement relationship is not one-one. Such relationships may need to take into account event ordering and iteration. The diagrams consist of a tree where nodes correspond to events; each level of the tree corresponds to a level of refinement, and concreteness increases with tree depth. The events of each refinement are read from left to right, at each level, and indicate the sequence in which the events are required to occur. The solid lines connecting events represent event refinement, and dashed lines represent events that refine skip, and add behaviour related to the abstract event. We indicate the parameter names of each event in the form of an event signature.

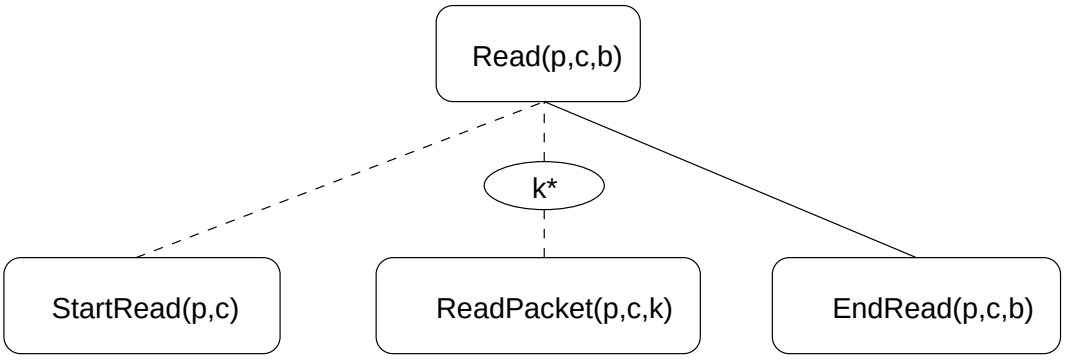
We can see from the diagrams of Figure 6.2 and 6.3 the top-level $Read$ and $Write$ events which are parameterized by a process p, channel c and block b. In the refinement we decompose the atomicity of the $Read$ and $Write$ events; the $Write$ event is decomposed to the three events, $StartWrite$, $WritePacket$ and $EndWrite$, of which the last refines $Write$. Similarly the $Read$ event is decomposed in to $StartRead$, $ReadPacket$ and $EndRead$ events. The refined events are parameterized by a process p, channel c, packet k, and data d.

In all Jackson Structure type Diagrams the '*' denotes possible iteration, where $k*$ indicates that the number of iterations is determined by a guard involving parameter k. In diagram 6.2 $k*$ indicates iteration for all packets k, where p is a process in $proc$ and c is a channel in $chan$; k is a packet in the buffer of p, $buff(p)$; and the packet is not already written to the channel c, $k \notin dom(data2(c))$. In our model all packets are eventually transferred to the channel buffer $data2(c)$ and iteration ceases. A similar scenario occurs in the read process.



where k* = ∀ k,p,c ·
p∈ proc ^ c∈ chan ^
k∈ dom(buff(p)) ^
k∉ dom(data2(c))

FIGURE 6.2: Decomposing the Write Event



where k* = ∀ k,p,c ·
p∈ proc ^ c∈ chan ^
k∈ dom(buff2(p)) ^
k∉ dom(data2(c))

FIGURE 6.3: Decomposing the Read Event

6.1.1 The Initial Event-B Model

At the highest level of abstraction we model processes, channels and data. We define carrier sets for the set of processes $Process$, the set of channels $Channel$, and set of data blocks represented by $Block$. A block of data is a function of packet identifiers to data, $Block = PKTID \rightarrow\!\!\!\!\!\!\rightarrow DATA$. Process objects are represented by a variable *proc*, and channels are represented by a variable *chan*. Each process has a local buffer called $buff$, and channels hold data in a buffer called *data*. Initially we model data transmission (reading from channel to process and writing from process to channel) as movement of an entire block of data. The intention is to model a system where one

block of data is copied to the channel, and a single reader copies the whole block to its local buffer; we are not modelling a system with multiple readers.

We define the variable types in the invariant as follows and initialize the variables as empty sets in the Initialisation clause,

INVARIANTS

$$proc \subseteq PROCESS$$
$$chan \subseteq CHANNEL$$
$$data \in chan \rightarrow Block$$
$$buff \in proc \rightarrow Block$$

The event models the write of a block of data b to a channel. The event parameters model the writing process p, and the target channel c. In addition to these parameters an additional local parameter b is introduced to keep track of the block of data to be written from the local buffer; this is defined as $b = buff(p)$. The event is guarded to ensure that the write only takes place if the buffer contains some data, i.e. $b \neq \varnothing$, and the channel buffer $data(c)$ is empty, i.e. $data(c) = \varnothing$.

$Write \triangleq$
ANY $p,\ c,\ b$
WHERE $p \in proc\ \wedge\ c \in chan\ \wedge b = buff(p) \wedge$
$buff(p) \neq \varnothing\ \wedge\ data(c) = \varnothing$
THEN $data(c) := b\ \parallel\ buff(p) := \varnothing$
END

The block b is copied to the data buffer $data(c)$ and the local buffer $buff(p)$ is emptied.

The event to read a block from the channel is parameterized by a reading process p, a channel c. In addition to these parameters we introduce the local parameter b to keep track of the block of data to be read from the channel buffer $data(c)$, it is defined as $b = data(c)$. The read event takes place only when the local buffer $buff(p)$ is empty, i.e. $buff(p) = \varnothing$; and the channel buffer has a block of data b to be read, i.e. $b \neq \varnothing$.

$Read \triangleq$
ANY $p,\ c,\ b$
WHERE $p \in proc\ \wedge\ c \in chan\ \wedge\ b = data(c)$
$buff(p) = \varnothing\ \wedge\ data(c) \neq \varnothing$
THEN $data(c) := \varnothing\ \parallel\ buff(p) := b$
END

The block in the channel buffer b is copied to the local buffer $buff(p)$ and the channel buffer $data(c)$ is emptied.

The event $NewProc$ adds processes from $PROCESS \setminus proc$ to the set $proc$, which is similar to the modelling of instantiation.

$$
\begin{array}{l}
NewProc \triangleq \\
\quad \textbf{ANY}\ p,\ b \\
\quad \textbf{WHERE}\ p \in PROCESS \setminus proc\ \wedge\ b \in Block \\
\quad \textbf{THEN}\ proc = proc \cup \{p\}\ \parallel\ buff(p) = b \\
\quad \textbf{END}
\end{array}
$$

The event $NewChan$ adds channels from $CHANNEL \setminus chan$ to the set $chan$,

$$
\begin{array}{l}
NewChan \triangleq \\
\quad \textbf{ANY}\ c \\
\quad \textbf{WHERE}\ c \in CHANNEL \setminus chan \\
\quad \textbf{THEN}\ chan = chan \cup \{c\}\ \parallel\ data(c) = \varnothing \\
\quad \textbf{END}
\end{array}
$$

6.1.2 Refinement with Data Packets

In the first refinement of the abstract machine we introduce writing behaviour which is performed in three steps, relating to the $StartWrite$, $WritePacket$, and $EndWrite$ events. Similarly $StartRead$, $ReadPacket$ and $EndRead$ perform reading. We record which activity (either reading or writing) a process may be engaged in using the variables $reading$ and $writing$. We also introduce $buff2$ which is a local buffer where data can be added or removed one packet at a time; and $data2$ which is a channel buffer where data can be added or removed one packet at a time. We type the additional variables of the machine as follows,

$$
\begin{array}{l}
\textbf{Invariants} \\
\quad writing \in proc \rightarrowtail\!\!\!\!\!\!+\ chan \\
\quad reading \in proc \rightarrowtail\!\!\!\!\!\!+\ chan \\
\quad buff2 \in proc \rightarrow Block \\
\quad data2 \in chan \rightarrow Block
\end{array}
$$

The typing invariant for $writing$ ensures that any process in the domain of $writing$ is linked to at most one channel in $chan$; and that the channel is related to, at most, one process. So only one process can write to a channel. Similarly, $reading$ ensures that

any process in the domain of *reading* is linked to at most one channel in *chan*; and that the channel is related to, at most, one process. So only one process can read from a channel. Each reading or writing process has a single local buffer $buff2$; any process in the domain of $buff2$ is related to a block of data of type *Block*. Each channel also has a single local buffer, $data2$, in which to store data; each channel in the domain of $data2$ is related to a block of data of type *Block*.

We ensure processes cannot be reading and writing at the same time with the invariant,

$$dom(writing) \cap dom(reading) = \varnothing$$

However we allow channels to be in the range of both *reading* and *writing* simultaneously.

We now look at the added events, firstly $StartWrite$ which refines skip. The event can occur when process p and channel c are not involved with writing and p is not reading. Additionally the local buffer $buff2(p)$ must have some data to transfer, i.e. $buff2(p) \neq \varnothing$; and the receiving channel buffer $data2(c)$ must be empty, i.e. $data2(c) = \varnothing$.

$$
\begin{array}{l}
StartWrite \triangleq \\
\quad \textbf{ANY } p,\ c \\
\quad \textbf{WHERE } p \in proc \ \wedge \ c \in chan \ \wedge \ p \notin dom(writing) \ \wedge \\
\qquad c \notin ran(writing) \ \wedge \ buff2(p) \neq \varnothing \ \wedge \ data2(c) = \varnothing \wedge \\
\qquad p \notin dom(reading) \\
\quad \textbf{THEN } writing := writing \cup \{p \mapsto c\} \\
\quad \textbf{END}
\end{array}
$$

The process and channel $p \mapsto c$ are added to the set of writing pairs.

Once a process-channel pair are added to the set of writing pairs we can transfer individual packets of data, represented by the maplet $k \mapsto d$, from one to the other. Here k represents the packet identifier and d represents the data. We introduce the $WritePacket$ event which refines skip to do this. The event will occur only when there is data to transfer out of the local buffer that is not already in the channel buffer, so it is guarded by,

$$k \in dom(buff2(p)) \ \wedge \ d = buff2(p)(k) \ \wedge \ k \notin dom(data2(c))$$

The channel buffer may or may not be empty. The $WritePacket$ event is defined in the following way,

$$
\begin{array}{l}
WritePacket \triangleq \\
\quad \textbf{ANY } p,\ c,\ k,\ d \\
\quad \textbf{WHERE } p \mapsto c \in writing \ \wedge \ k \in dom(buff2(p)) \ \wedge \\
\qquad d = buff2(p)(k) \ \wedge \ k \notin dom(data2(c)) \\
\quad \textbf{THEN } data2(c) := data2(c) \cup \{k \mapsto d\} \\
\quad \textbf{END}
\end{array}
$$

In the $WritePacket$ event a packet $k \mapsto d$ is added to the channel buffer $data2(c)$.

The $EndWrite$ event refines $Write$. The write activity ends when the channel buffer is equal to the local buffer, $data2(c) = buff2(p)$. We introduce a witness using the **WITH** clause to link the abstract block of data in the buffer b, to the refined buffer $buff2$.

$$
\begin{array}{l}
EndWrite \triangleq \\
\quad \textbf{REFINES } Write \\
\quad \textbf{ANY } p,\ c \\
\quad \textbf{WHERE } p \mapsto c \in writing \ \wedge \ c \in chan \ \wedge \ data2(c) = buff2(p) \\
\quad \textbf{WITH } b = buff2(p) \\
\quad \textbf{THEN } writing := \{p\} \lhd writing \parallel buff2(p) := \varnothing \\
\quad \textbf{END}
\end{array}
$$

The process p is removed from the set of writers $writing := \{p\} \lhd writing$ and the local buffer is cleared, $buff2(p) := \varnothing$.

The read activity begins with a $StartRead$ which refines skip. It can occur when process p and channel c are not involved with reading, $p \notin dom(reading) \ \wedge \ c \notin ran(reading)$; and p is not writing, $p \notin dom(writing)$.

$$
\begin{array}{l}
StartRead \triangleq \\
\quad \textbf{ANY } p,\ c \\
\quad \textbf{WHERE } p \in proc \ \wedge \ c \in chan \ \wedge \ p \notin dom(reading) \ \wedge \\
\qquad c \notin ran(reading) \ \wedge \ p \notin dom(writing) \ \wedge \ data2(c) \neq \varnothing \ \wedge \ buff2(p) = \varnothing \\
\quad \textbf{THEN } reading \cup \{p \mapsto c\} \\
\quad \textbf{END}
\end{array}
$$

The process and channel pair $p \mapsto c$ are added to the set of reading pairs.

The reading of individual packets, represented by the maplet $k \mapsto data2(c)(k)$, can occur as soon as a packet appears in the channel buffer, $k \in dom(data2(c))$. Here k is a packet

identifier and $data2(c)(k)$ relates to the data. The event can transfer data as long as it is not already in the local buffer; it is guarded by $k \notin dom(buff2(p))$. The event *ReadPacket* refines skip, and has a process p and channel c. A process can only read from a channel when it is in the set of reading pairs denoted by $p \mapsto c \in reading$.

$ReadPacket \triangleq$
ANY $p,\ c,\ k$
WHERE $p \mapsto c \in reading\ \wedge\ k \in dom(data2(c))\ \wedge\ k \notin dom(buff2(p))$
THEN $buff2(p) := buff2(p) \cup \{k \mapsto data2(c)(k)\}$
END

A packet from the channel buffer, represented by $k \mapsto data2(c)(k)$, is added to the local buffer $buff2(p)$.

Reading of packets ends when the channel buffer is equal to the local buffer, $buff2(p) = data2(c)$; and the channel is no longer being written to, represented by the guard $c \notin ran(writing)$. We introduce a witness using the **WITH** clause to link the block of data b in the abstract event to the channel buffer $data2(c)$. The *EndRead* event refines *Read* and is defined as follows,

$EndRead \triangleq$
REFINES *Read*
ANY $p,\ c$
WHERE $p \in proc\ \wedge\ c \in chan\ \wedge p \mapsto c \in reading\ \wedge$
$c \notin ran(writing)\ \wedge\ buff2(p) = data2(c)$
WITH $b = data2(c)$
THEN $data2(c) := \varnothing\ \parallel\ reading := reading \setminus \{p \mapsto c\}$
END

The channel buffer is emptied in the event action, $data2(c) := \varnothing$, the data is considered to have been consumed by the reading process. The process-channel pair is removed from the set of reading pairs, $reading := reading \setminus \{p \mapsto c\}$.

During the refinement process a number of gluing invariants were added. In order to show that the channel data block *data* is equal to the packetized data *data*2, except when the process is writing, we have the following invariant,

$$\forall c \cdot c \in chan \wedge c \notin ran(writing) \Rightarrow data(c) = data2(c)$$

We have a similar invariant to show that the process block buffer $buff$ must be the same as the packetized buffer $buff2$, except when the process is reading.

$$\forall p \cdot p \in proc \wedge p \notin dom(reading) \Rightarrow buff(p) = buff2(p)$$

The refinement of the abstract development is now complete, and we have been able to discharge all proof obligations that have arisen. We are now ready to begin the implementation stage of development.

6.1.3 The OCB Specification

When we are making design decisions about the implementation of the reading and writing processes we made use of the diagrams of Figures 6.2 and 6.3 to help visualise the required implementation. For example, from the diagrams we can see that the write behaviour needs to be implemented as a sequence of atomic steps, and one of the steps, *writePacket*, is iterative so the use of the looping construct will be required. In the implementation level diagram we use the OCB clause labels to describe the atomic events, which can be seen in Figures 6.4 and 6.5. In the implementation level diagram we indicate iteration of a group of clauses by attaching the loop condition annotation to an enclosing box. In our OCB specification we implement the events of the abstract development with one or more OCB labelled atomic clauses, one of which explicitly refines an abstract event (the others refine SKIP).

In the OCB refinement of the abstract development, design decisions need to be taken to define the relationship between the OCB process class specification and the abstraction needs to be defined. We know that the read and write behaviours are mutually exclusive, that is a process will either read or write, but not both. We could implement the processes as two separate classes embodying the separate behaviours. Alternatively we can implement one process class specification that can do either task, and we differentiate between readers and writers by supplying a parameter at the time of invocation. It is the latter approach that we choose for our implementation. The only construct that is shared between processes is the channel, we specify the channel as a monitor class, but is not apparent from the Jackson Structure type Diagrams since they describe behavioural aspects of the system and monitor classes play a passive role in the system. Figures 6.4 and 6.5 show the relationship between the events of the abstract development to the corresponding clauses of the OCB specification; clause labels are used instead of event names, in the diagram to describe the implementation level constructs. Iteration of clauses, at the implementation level, are grouped by an enclosing box and the labelled annotation show the loop condition.

We can see that the $StartWrite$ event is implemented by clauses labelled $p1$ and $p2$ which make preparations for writing by obtaining the size of the local buffer and obtaining

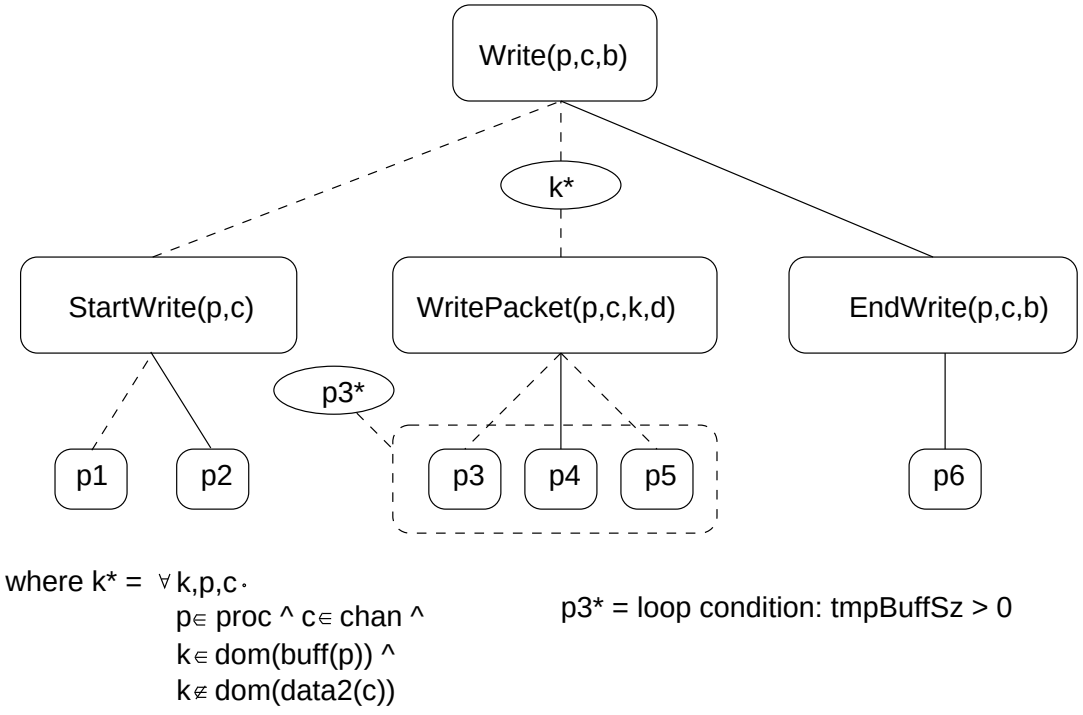


FIGURE 6.4: Decomposing the Write Event - at the Level of OCB Specification.

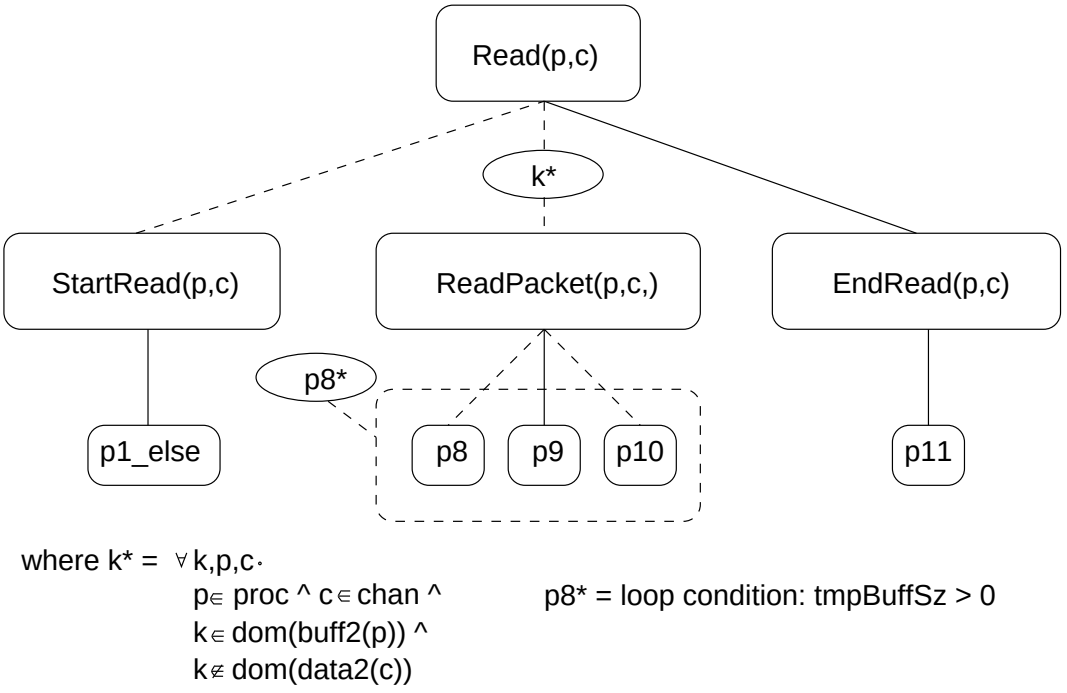


FIGURE 6.5: Decomposing the Read Event - at the Level of OCB Specification.

the write channel if it is free (and blocking otherwise). In the abstraction a channel is restricted to being written to by one process. This is embodied in the guard $c \notin ran(writing)$ of the $StartWrite$ event. However in the implementation we do not have access to set constructs so we introduce an attribute $wPID$ to the shared channel class. Each process will have a unique identifier and it is assigned to $wPID$ when the process is writing to the channel, or otherwise has the value -1 if no process is writing to it. In this way only a single process/channel pair are linked for the purposes of writing, and since the $getWChan$ procedure is guarded by the **when** clause **when**$(wPID = -1 \ldots$, only a single writing process can be associated with the channel at any one time. In the generated Event-B model a guard causes the event to block if $\neg wPID = -1$. So the blocking behaviour described by OCB's **when** construct will cause a process to wait when attempting to obtain a channel which is not available. We also use this approach when a reader tries to request data from a channel but there is none in the buffer, yet the write is not complete. The $freeWChan$ procedure then resets the $wPID$ attribute to the value of -1 when the write has finished.

Now we return to the discussion about processes and refer again to Figure 6.4 - The clause $p2$ obtains mutually exclusive access to the channel by calling the $getWChan$ procedure, which directly refines the $StartWrite$ event. In the abstraction the $WritePacket$ event then iterates over the packets of data in the buffer. We implement this as a while loop with labelled clauses $p3 \ldots p5$. Following this clause $p6$ releases the channel by calling the $freeWChan$, so it directly refines the $EndWrite$ event. A description of the write process behaviour is summarized in the following table, which also includes the reading process description.

Label	Description
p1	If the process is a writer then get the size of the local buffer.
p2	Obtain the write channel if it is free else block.
	The process ID and number of packets to send are parameters
p3	While there is a packet to send from the local buffer
	remove the packet assigning to the temporary attribute.
p4	Add the data to the channel buffer.
p5	Decrement the count of packets.
p6	Release the channel for other writers.
p1_else	If the process is a reader obtain a read channel
	if it is free, else block.
p7	Obtain the number of packets to read from the channel.
p8	While there are packets remove a packet from the channel
	buffer and assign to the temporary attribute.
p9	Add the packet to the local buffer.
p10	Decrement the buffer counter.
p11	Free the read channel for another reader.

We can see in the diagram that the labelled atomic clauses $p2 \ldots p6$ ultimately implement the most abstract $Write$ event, and we can also see that clauses $p3 \ldots p5$ refine the $WritePacket$ event. Each of the events $StartWrite$, $WritePacket$ and $EndWrite$ are refined by exactly one labelled OCB clause, along with a number of associated labelled clauses. The additional associated clauses contain loop control information, and manipulate the data. For example the loop $p3 \ldots p5$ relies on keeping track of the number of packets of data to write. The number of elements to write is initially obtained in clause $p1$ by calling $buff$'s $getSize$ procedure - only clause $p4$ refines $WritePacket$ explicitly which involves the write of a packet of data to the channel. This can be seen in the following $Proc$ process class specification.

In our implementation we specify reading and writing processes in the same class, the behaviour of the process is determined by the boolean parameter $isWriter$ supplied at instantiation. The process class $Proc$ specification follows,

```
ProcessClass Proc{
 Buffer buff, Boolean isWriter, Channel c, Integer id,
 Integer tmpBuffSz, Integer tmpDat
 // The constructor procedure
 Procedure create(Integer pid, Buffer bff,
                      Boolean isWritr, Channel ch){
  id:=pid || buff:=bff || isWriter:=isWritr || c:= ch ||
  tmpBuffSz:=-1 || tmpDat:=-1
```

```
 }
 // The process behaviour
 Operation run(){
  p1: if(isWriter=TRUE) then
       tmpBuffSz:=buff.getSize() andthen
       p2: c.getWChan(id, tmpBuffSz);  // refines StartWrite
       p3: while(tmpBuffSz>0) do tmpDat:=buff.remove() andthen
            p4: c.add(tmpDat);  // refines WritePacket
            p5: tmpBuffSz:=tmpBuffSz-1 endwhile ;
       p6: c.freeWChan() endif  // refines EndWrite
  else c.getRChan(id) andthen  // refines StartRead
       p7: tmpBuffSz:=c.getWriteSize();
       p8: while(tmpBuffSz>0) do tmpDat:=c.remove() andthen
            p9: buff.add(tmpDat);  // refines ReadPacket
            p10: tmpBuffSz:=tmpBuffSz-1 endwhile ;
       p11: c.freeRChan() endelse  // refines EndRead
 }
}
```

The process class constructor parameters include a process identifier pid, an internal buffer bff, and the shared channel ch. There are additional attributes for temporary storage of values used during processing. We have now specified the $Proc$ process class and $Channel$ monitor class that we need to implement the system. There is however another class, the local Buffer class, used within the process class. It is very similar to that of the channel class, with add and remove procedures for which we provide no further details here (see Appendix C.4 for details)

We next discuss some more aspects of the monitor class specification of the Channel. The OCB $Channel$ specification has an array buffer $buff$ of, capacity 50; integer data elements are added to the write location $wLoc$ and read from the read location $rLoc$ of the buffer. The add procedure limits bursts of data to 5 items using the guard $wLoc - rLoc \leq 5$ in the $when$ clause, but can only transmit up to 50 elements in total due to the array size. The attributes $rPID$, $wPID$ and $writeSize$ record the identifier of the reading process, writing process, and size of block to be written respectively. $rPID$ and $wPID$ are initially assigned a value of -1 which we will not use for a process identifier. There are seven monitor procedures for which we summarize in the following table,

add	Add a packet to the buffer at the write location.
	Block if the write limit is reached
remove	Remove and return a packet from the buffer at the read.
	Block the caller if there is nothing to remove.
getWChannel	Obtain a channel for writing if it is available
	and there is no reader, else block the caller.
freeWChan	Release a write channel by removing the process ID.
getRChannel	Obtain a channel for reading if it is available
	and there is data to read, else block the caller.
freeRChan	Release a read channel by removing the process ID.
getWriteSize	Returns the size of the data block.

We reproduce the Channel specification here for reference,

```
MonitorClass Channel{
 // Attributes
 Integer[50] buff, Integer rLoc, Integer wLoc,
 Integer rPID, Integer wPID, Integer writeSize

 // The Constructor
 Procedure create(){
  rLoc:= 0 || wLoc:= 0 || rPID:= -1 ||
  wPID:= -1 || writeSize:= -1
 }

 // 'Refines' WritePacket - in a call from clause p4
 Procedure add(Integer val){
  when(wLoc - rLoc <= 5){
        buff[wLoc]:= val || wLoc:= wLoc + 1}
 }

 // The value is stored in a temporary buffer in a
 // call from clause p8 - implementing ReadPacket
 //as part of the reading activity.
 Procedure remove(){
  when(wLoc - rLoc > 0){
        return:= buff[rLoc] || rLoc := rLoc+1 }
 }: Integer

 // Called in p1_else clause - refines StartRead.
 // Set the channel for reading, by the process
 // with identifier pid.
```

```
// Block if it is already owned or has nothing to read.
Procedure getRChan(Integer pid){
  when(rPID=-1 & writeSize>0){rPID:= pid}
}

// Called in p11 clause - refines EndRead.
// Free the channel for reading.
Procedure freeRChan(){
  rPID:= -1 || writeSize:= -1
}

// Called in p1 clause - implementing StartWrite.
// Set the channel for writing writesze bytes, by
// the process pid.
// Block if the channel is already owned for writing or
// has bytes still to write.
Procedure getWChan(Integer pid,Integer writeSze){
  when(wPID=-1 & writeSize<=0){
        wPID:= pid || writeSize:= writeSze}
}

// Called in p6 clause - refines EndWrite.
// Free the channel for writing.
Procedure freeWChan(){ wPID:= -1 }

// Return the number of bytes to write.
Procedure getWriteSize(){ return:= writeSize }: Integer
}
```

A *MainClass* is specified in Appendix C.3, with a *main* operation, which is used as the entry point for execution. In clauses labelled $m1 \ldots m4$ local buffers are constructed for use by four processes. Two of these processes will be reader processes and two will be writer processes, which is determined by supplying the appropriate constructor parameter. In $m5$ a channel is created. In $m6 \ldots m8$ the buffers used for writing are filled with some arbitrary data. Clauses $m9 \ldots m12$ are used to create the four processes.

6.2 The Event-B Model of the OCB Specification

The Event-B model arising from the OCB specification is too large to discuss in detail in its entirety; we focus on the *Proc* class' write activity specified in clauses $p1 \ldots p6$ to

provide an overview, more details are available in Appendix C.8. $Proc\_p1$ is the event that arises from the clause labelled $p1$,

$$
\begin{array}{l}
Proc\_p1 \triangleq \\
\quad \textbf{ANY } self,\ target \\
\quad \textbf{WHERE } self \in Proc \ \wedge \ self \in dom(Proc\_state) \ \wedge \\
\qquad Proc\_state(self) = p1 \ \wedge \ Proc\_isWriter(self) = TRUE \ \wedge \\
\qquad self \in dom(Proc\_buff) \ \wedge \ target = Proc\_buff(self) \\
\quad \textbf{THEN } Proc\_tmpBuffSz(self) := \\
\qquad Buffer\_wLoc(target) - Buffer\_rLoc(target) \ \| \\
\qquad Proc\_state(self) := p2 \\
\quad \textbf{END}
\end{array}
$$

$Proc\_state(self) = p1$ is the guard relating to the program counter, the guard relating to the **when** clause is $Proc\_isWriter(self) = TRUE$, and identification of the target of the procedure call is $target = Proc\_buff(self)$. In the event action $Proc\_state(self) := p2$ the program counter is advanced. The $getSize$ procedure call is expanded to create the assignment,

$$Proc\_tmpBuffSz(self) := Buffer\_wLoc(target) - Buffer\_rLoc(target)$$

The event arising from the clause $p2$ obtains the channel's buffer for writing, or else blocks. This is embodied in the guard $Channel\_wPID(target) = -1$. The process also blocks if data in the channel buffer is still being read, which is associated with the guard $Channel\_writeSize(target) \leq 0$. The value of
$Channel\_writeSize(target)$ is reset when a reader frees the channel using $freeRChan$, and similarly the value of $Channel\_wPID(target)$ is reset when a writer frees a channel using $freeWChan$.

$Proc\_p2$ refines $StartWrite$ of the first refinement. The $StartWrite$ process p is related to $self$ of $Proc\_p2$ using a predicate $p = self$ in the event's $WITH$ clause. Relating the OCB specification to the abstract development, and how it can best incorporated into the approach, is the subject of future work; for now we just state the relationships.

The $StartWrite$ channel c is related to $target$ of $Proc\_p2$ with $c = target$.

$$
\begin{array}{l}
Proc\_p2 \triangleq \\
\quad \textbf{REFINES } StartWrite \\
\quad \textbf{ANY } self,\ target \\
\quad \textbf{WHERE } self \in Proc \ \wedge \ self \in dom(Proc\_state) \ \wedge \\
\qquad Proc\_state(self) = p2 \ \wedge \ self \in dom(Proc\_c) \ \wedge \\
\qquad target = Proc\_c(self) \ \wedge \ Channel\_wPID(target) = -1 \ \wedge \\
\qquad Channel\_writeSize(target) \leq 0 \\
\quad \textbf{THEN } Channel\_wPID(target) := Proc\_id(self) \ \wedge \\
\qquad Channel\_writeSize(target) := Proc\_tmpBuffSz(self) \\
\qquad Proc\_state(self) := p3 \\
\quad \textbf{END}
\end{array}
$$

In the event action the process identifier is supplied to the channel as the writer identifier, $Channel\_wPID(target) := Proc\_id(self)$. The number of packets to write is set in the channel's $writeSize$ attribute using,

$$Channel\_writeSize(target) := Proc\_tmpBuffSz(self)$$

and the program counter is updated to the next value.

In the event $Proc\_p3$ repeating behaviour begins, a packet of data is removed from the local buffer and stored for insertion into the channel buffer in the next clause. The condition of the OCB while clause gives rise to the guard $Proc\_tmpBuffSz(self) > 0$; other guards arise from the conditional waiting clause relating to the $remove$ procedure call - this is the same as channel's $remove$ procedure (since the buffers are the same size). The process is required to wait if there is no data to remove from the channel, progress is enabled therefore by the guard $Buffer\_wLoc(target)Buffer\_rLoc(target) > 0$. Other

guards are added to assist with discharging the well-definedness proof obligations associated with partial functions, such as $self \in dom(Proc\_buff)$.

$$
\begin{array}{l}
Proc\_while\_p3 \triangleq \\
\quad \textbf{ANY } self,\ target \\
\quad \textbf{WHERE } self \in Proc \ \wedge \ self \in dom(Proc\_state) \ \wedge \\
\qquad Proc\_state(self) = p3 \ \wedge \ Proc\_tmpBuffSz(self) > 0 \ \wedge \\
\qquad self \in dom(Proc\_buff) \ \wedge \ target = Proc\_buff(self) \ \wedge \\
\qquad Buffer\_wLoc(target) - Buffer\_rLoc(target) > 0 \\
\quad \textbf{THEN } Proc\_tmpDat(self) := \\
\qquad\qquad Buffer\_buff(target)(Buffer\_rLoc(target)) \parallel \\
\qquad Buffer\_rLoc(target) := Buffer\_rLoc(target) + 1 \parallel \\
\qquad Proc\_state(self) := p4 \\
\quad \textbf{END}
\end{array}
$$

In the following fragment the value at the read location rLoc of the buffer is assigned to an attribute,

$$Proc\_tmpDat(self) := Buffer\_buff(target)(Buffer\_rLoc(target))$$

the buffer read location is then incremented,

$$Buffer\_rLoc(target) := Buffer\_rLoc(target) + 1$$

The details shown here have given an overview of how the OCB specification maps to Event-B. We discuss briefly the refinement of the $WritePacket$ and $EndWrite$ events, but with less detail. $WritePacket$ is refined by $Proc\_p4$, which models the addition of packets to the channel buffer using *Channel*'s *add* procedure. The greatest complexity with this part of the mapping is the relational override associated with an array update. The *add* procedure contains the clause $buff[wLoc] := val$ which specifies the addition of the packet val to the channel buffer at the $wLoc$ index.

$$
\begin{array}{l}
Channel\_buff(target) := \\
\quad Channel\_buff(target) \lhd\!\!+ \{Channel\_wLoc(target) \mapsto Proc\_tmpDat(self)\}
\end{array}
$$

In the mapping the OCB formal parameter val, is substituted with the actual parameter, $Proc\_tmpDat(self)$. $Channel\_wLoc(target)$ corresponds to the index value $wLoc$. The guard $Channel\_wLoc(target) - Channel\_rLoc(target) \leq 5$ causes the writing process to block if the channel buffer has 5 elements and allows the reading to commence. In the action of the event $Proc\_p5$ the count of packets in the buffer is decremented.

$$Proc\_tmpBuffSz(self) := Proc\_tmpBuffSz(self) - 1$$

EndWrite is refined by $Proc\_p6$, we indicate that the process is no longer in the writing set by setting the channel's $wPID$ to -1 in the $freeWChan$ procedure; no process will have this identifier. The event action for the mapping is $Channel\_wPID(target) := -1$.

When proving the refinement of the implementation model we use gluing invariants to relate the abstraction with the implementation. An example of such an invariant follows,

$$\forall self \cdot self \in Proc \land self \in dom(Proc\_id) => Proc\_id(self) \geq 0$$

We include the invariant above because we wish to ensure that no processes have the identifier -1, which is reserved for indicating that the channel has no reader/writer. We then have an invariant that states that if a channel does not have a writing process identifier value as its $wPID$ attribute (since Channel_wPID(Proc_c(self)) = -1) then this implies that the process is not in the domain of the writing set in the abstraction.

$$\begin{aligned}
&\forall self \cdot self \in Proc \land \\
&self \in dom(Proc_c) \land \\
&Channel_wPID(Proc_c(self)) = -1 \\
&\Rightarrow self \notin dom(writing)
\end{aligned}$$

We also relate the channel to the abstract writing set,

$$\begin{aligned}
&\forall self \cdot self \in Proc \land \\
&self \in dom(Proc_c) \land \\
&Channel_wPID(Proc_c(self)) = -1 \\
&\Rightarrow Proc_c(self) \notin ran(writing)
\end{aligned}$$

Similar invariants exists for the readers.

We state that all process identifiers should be unique in the following invariant:

$$\begin{aligned}
&\forall p, q \cdot p \in Proc \land q \in Proc \land \\
&p \neq q \\
&\Rightarrow Proc_id(p) \neq Proc_id(q)
\end{aligned}$$

We ensure that the write location $wLoc$ remains in the bounds of the array with the invariant:

$$\begin{aligned}&\forall self \cdot self \in Proc \land \\ &self \in dom(Proc\_buff) \\ &\Rightarrow Buffer\_rLoc(Proc\_buff(self)) \in 0 \mathinner{..} 49\end{aligned}$$

To assist with the proof we added the following theorem. It is a form of re-use strategy; the predicate was frequently added to the hypothesis and used to discharge proof obligations. By adding it as a theorem it becomes available to the automatic proof tools.

$$(\lambda i \cdot i \in 0 \mathinner{..} 49) \in (0 \mathinner{..} 49 \rightarrow \mathbb{Z})$$

6.3 The Java Implementation

The mapping to Java is mostly self evident since it is very similar to the OCB specification. We present the source code for the Channels Class, Proc and CommBuffer respectively in Appendices C.5,C.6,C.7. The *remove* method of the channel class C.5 shows the conditional waiting mapping, and the temporary initial variables required to provide parallel semantics. We also see that all methods in the Channel are synchronized and those that may update state also call $notifyAll$ to wake waiting threads. The *main* method of the $CommBuffer$ class shows the creation and starting of the new threads.

6.4 Issues Arising from the Case Study

The motivation for the case study was to provide an implementation of a channel buffering system. We initially described the development using an abstract model which models transfer of one block of data at a time. We then refined the model using the transfer of the packets that make up a block. Then we presented a systematic approach to linking the refined model with an OCB specification which resulted in a translation to an Event-B model and Java code. We found the Jackson Structure type Diagrams were a useful aid to visualising the relationships between the abstract events and the events of the refinement. The Jackson Structure type Diagrams embody both sequencing and iteration, and we see how this links to sequence and looping in the OCB specification.

The OCB specification consists of a $MainClass$, one process class, and two monitor classes. There are 23 labelled clauses in total. These give rise to 41 events and 36 typing invariants in the Event-B model. The number of proof obligations generated from just these (before proving refinement) totalled 330, 300 of which were discharged immediately by the auto-prover, and the remainder were discharged relatively quickly in the

interactive proof environment. The proof of OCB refinement has not yet been completed, but all the proof obligations of the abstract development have been discharged. With regard to the refinement of an abstract development using OCB, we recognised some opportunities to incorporate productivity enhancements - such as adding refines and witness clauses to OCB operations. This will enable developers to indicate, during the specification stage, the relationship between OCB operations and the abstract events that they refine. Let us take a brief look at witnesses, which are useful when discharging proof obligations since they relate parameters of the abstract event to the refined event. Parameters of an abstract event do not have to appear in an event that refines it, but if they do not appear they can be linked to some variable, or parameter, in the refinement using a witness clause. In our example the $StartWrite$ event is refined by $Proc\_p2$ event. In the abstraction we have local variables p and c where $p \in proc \wedge c \in chan$. We link these to concrete objects in the OCB refinement using witnesses $p = self \wedge c = Proc\_c(self)$. A refines clause could be associated with a labelled OCB clause to indicate which abstract event it refines. The translation from the OCB specification to Java code worked as expected, and we were able to run the program in the JVM.

6.4.1 Tooling Issues

In this section I describe how progress was hampered by some tooling issues which prevented proof of refinement. We then describe one possible solution to overcome some of the problems. The implementation level model contains over 600 proof obligations, 60 Invariants and 40 events. The majority of these proof obligations have been discharged with the addition of new gluing invariants; over 250 automatically and over 250 using the interactive prover. There remain 40 or so (non-trivial) proof obligations to discharge. However as modelling progressed it became clear that the tool was having difficulty with the relatively large model. Throughout our investigations we have found that the size of Event-B models, associated with OCB models, is large and complex when compared to more abstract models. This results in extremely slow interaction with the proof tools. The tool's responsiveness to changes made by the user has made it difficult to finish proof of refinement in a reasonable period of time; and also various technical problems have hampered progress. In the remainder of this text we describe some of the problems encountered when working with larger, more complex models and give details of the responsiveness of the system. We then propose a solution which involves decomposition of the models into smaller parts. The solution that we propose will also serve to reduce the abstraction gap between the abstract Event-B development and implementation.

The use of implementation level models exposes several Rodin tool problems. For instance when the model is rebuilding Java out-of-memory errors are quite often encountered. To overcome this we can invoke the prover manually, on a smaller number of

proofs, but this slows the proof activity again and train of thought can be broken. Occasionally the database does not remain synchronized with the user interface; this gives rise to the proof appearing to be discharged in one pane, but not in another. Sometimes the prover enters a loop and does not exit; this leaves the database in an inconsistent state, and the model must be cleaned to restore the integrity. This gives rise to a loss of the recent proof activity, a loss of time, and disruption in the process of reasoning. On occasions, when trying to access a proof obligation by clicking on it, an exception is thrown stating that a proof attempt already exists. This again seems to be revealing inconsistent state in the Rodin database when compared to the user interface. These problems seem to manifest themselves more often in larger models; experience indicates that smaller models give rise to fewer problems. This is one of the reasons (but by no means the only reason) for proposing an approach involving decomposition of an implementation level model into smaller parts.

Partitioning of the models will improve responsiveness of the tool, i.e. the time taken to react to changes made by the user. We have measured the response time of the system by measuring the time taken to re-build the model, and attempt to automatically discharge the proof obligations, as a result of adding an invariant. In table 6.1 we can see a comparison between the time taken for versions 0.9.2 and 1.1 of the Rodin tool. The OCB Event-B translator is compatible with Version 0.9.2; The translation tool is not compatible with version 1.1, but we are able to import a model for comparison of performance. During the test we invoke an external prover known an the Mono-Lemma (ML) prover, a legacy component, and compare the time taken without the ML prover in both versions of the Rodin tool. There is a trade off in using the ML prover since it automatically discharges a large number of proofs, but it is slow. The data in

Rodin version	with ML	time (mins)
0.9.2	no	2.5
0.9.2	yes	2.75
1.1	no	1.5
1.1	yes	4

TABLE 6.1: Time to re-build a model

Table 6.1 indicates that there have been improvements in the performance of the proof tool, between versions 0.9.2 and 1.1, when used without the legacy ML component. So although a wait of 1.5 minutes is still quite a long time for a response to an update, it is an improvement on the older version of the tool. The data also suggests that the use of the ML prover is much less appropriate in the newer version of the tool. This increase may be due to an internal timeout having a different value in the two tool versions, rather than being a performance issue. The ML tool will eventually be replaced by a rule based prover of [107], which we expect to be more efficient. In summary we expect further performance gains to be made as tool development proceeds, so larger models will be more tractable. However we still propose that partitioning of the system will

bring other benefits, such as modular reasoning, and we continue with a description of out proposal. Our experience of OCB tool development, use of the tool and proof, and an overview of our proposed solution will be presented in [12].

6.4.2 Decomposition in Event-B with a View to Automatic Code Generation

We now propose our modular approach for using shared event decomposition to overcome the problems of models described in the previous section. Event-B modelling of concurrent systems sharing data, near the implementation level, can give rise to a model with considerable complexity. By splitting the specification we should be able to deal with more tractable partitions of the system. Our proposal is to make OCB an extension of the Event-B approach (with Event-B semantics). An abstract development can be decomposed using refinement into $ProcessMachines$ and $SharedMachines$. By using a decomposition approach we allow for a number of refinement levels; instead of the single level of an OCB specification. In doing so we reduce the abstraction gaps between the abstract development and implementation. The proposed Event-B extension performs a similar role to OCB and many of the OCB elements, such as sequence, looping, procedures, procedure calls, and so, on are used. In this work we will therefore retain the name OCB to describe the extension. Following the decomposition of an abstract development into a number of machines (which may be independently refined further) we can use the features of OCB to map the decomposed machines to the target implementation. We will have maintained the refinement link using the Event-B extension using its Event-B semantics; there will also be an underlying Event-B model which may be the result of a translation, or may be synchronized with the OCB model using Eclipse tools. The final translation to Java would be done using the approach described in the main body of the thesis. In our extension $ProcessMachines$ are implemented by threads, and $SharedMachines$ are implemented as monitors. A $SharedMachine$ specification is simply an Event-B machine, as in the current Rodin tool; this machine can the be further refined independently. A $ProcessMachine$ can refer to $SharedMachine$s and call their procedures. The procedure call will be modelled using the shared event decomposition approach of [33]. So a procedure call is modelled as a shared event where the events synchronize using their guards; guards also define the values of the input and output parameters. Figure 6.6 shows how the abstract development can be linked to the implementation through the decomposed machines.

We now provide details of the proposed new Event-B constructs, and the shared event approach. To accommodate decomposition into process machines and shared machines in Event-B we extend Event-B to allow a process to call a $SharedMachine$'s procedure, and denote such a call using the *call* keyword. A process machine T may call shared

machine M's event $Evt2$ using a *call* clause as follows:

Process Machine T
Variables $v, x, y,$
$\ldots$
Events
process $Evt2 \triangleq$
$x :| G1;$
$y \leftarrow$ **call** $M.Evt2(x);$
A
$\ldots$

The extended Event-B annotation permits the restricted use of sequence clauses in an event. One of the problems with the approach as it exists is that there are more atomic clauses than are strictly necessary. These give rise to more events (and therefore more complexity) than are strictly necessary. We have identified a frequently used pattern that can be used to reduce the number of events generated. In the above clause a non-deterministic assignment $x :| G1$ provides a value for x satisfying guard $G1$. Note that x is local to the process machine T, but is shared via a parameter in the call clause. The clause **call** $M.Evt2(x)$ calls a shared machine's event, and the result is assigned to the variable y atomically. Some remaining local processing may be undertaken in clause A, each variable's value may only be assigned once in each event. This is because we

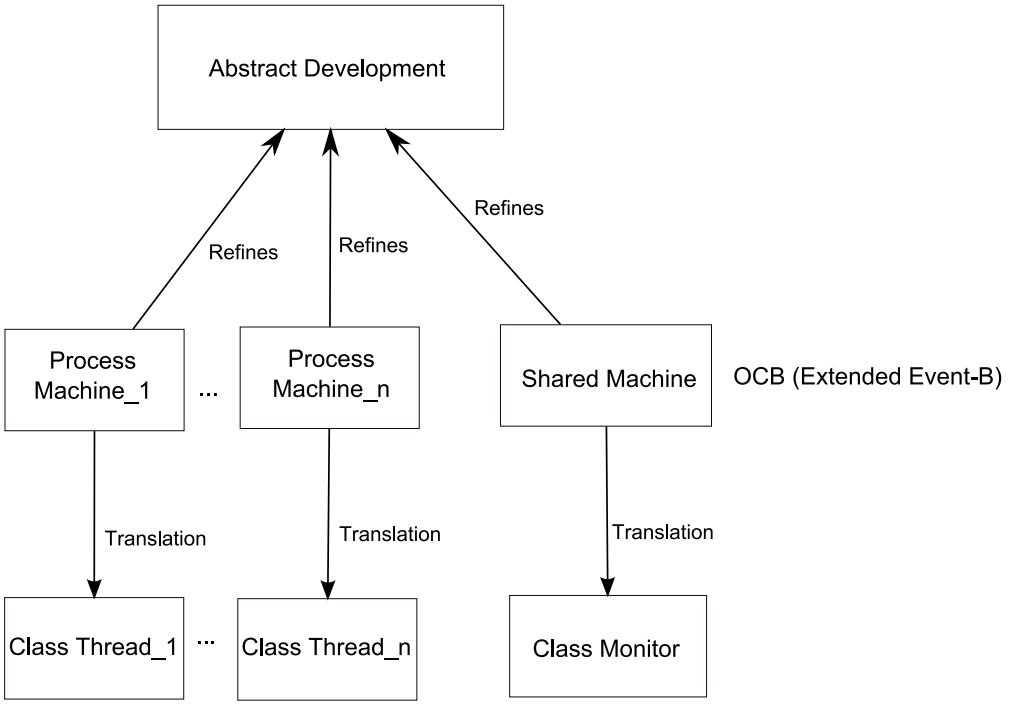


FIGURE 6.6: Decomposition of an Abstract Development

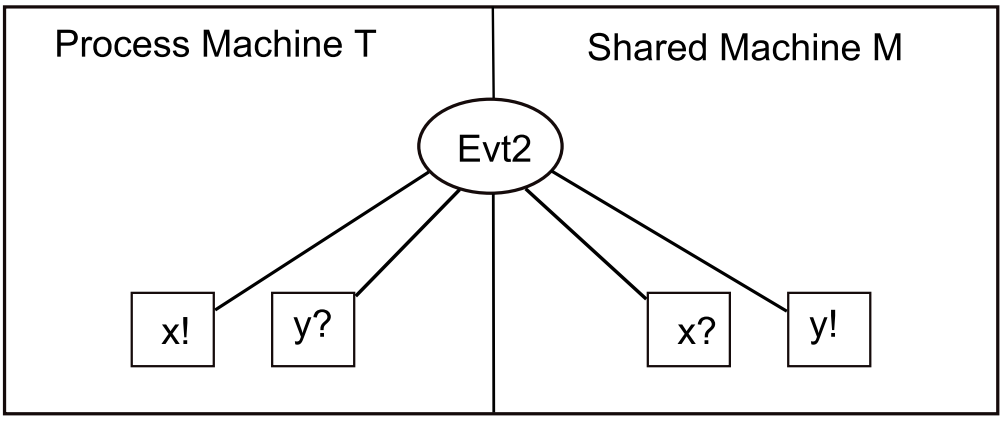


FIGURE 6.7: A Shared Event Refinement

wish to simplify the proof effort and do not wish to introduce a sequence operator to the underlying Event-B syntax for this solution.

The extended Event-B model has a representation of a $ProcessMachine$ in the underlying Event-B model. That is, the implementation level model will be different to its underlying representation, as is the case for the approach described in the main body of the thesis. However a $SharedMachine$ will appear as a normal machine in the implementation level specification; and this will include an extension to the decomposition approach. The extended decomposition approach will make use of annotations defining input and output parameters for events. We denote an input parameter x using the annotation $x?$, and an output parameter y as $y!$. The input and output values of the variables are shown in Figure 6.7.

Using this approach we hope to decompose the model into tractable partitions, and provide a more intuitive link between Event-B and the implementation. The extended Event-B will have subsumed OCB and will be used to specify instantiation, and ordering of events. It will also allow us to specify points for interleaving, as before, in non-atomic constructs. The partitioning into smaller models should give rise to fewer problems when using the tools. In addition to this smaller models are easier to reason about, and machines can be refined independently.

Due to constraints applied to our approach we are limited in the form of implementation; and we will investigate further, in an extension to this work, a transactional approach. We propose an approach to facilitate access to a number of objects directly. Currently the user interface is in need of some improvement, and much could be done to smooth the development process on this front.

Chapter 7

Case Study 2

In this chapter we present a case study arising from investigations into the modelling of the Intel Flash File System Core Reference Guide [82], which used as a reference document for the Verification Grand Challenge [66]. We show how OCB can be used to implement part of the Flash File System API layer of the Flash File System Core Reference. A formal presentation of a tree structured file system is presented in [48], but our focus is at a higher level in the hierarchical specification, closer to the user interface. We present a systematic approach to linking the refined model with an OCB specification which results in a translation to an Event-B model and Java code. This case study highlights a significant limitation of the approach that we have advocated thus far, and incentivises further work introduced in Chapter 8. We begin with an overview of the Flash File System in question, giving details of the parts that we will model and implement.

7.1 The Flash File System Core

During this work we will refer to the Intel Flash File System Core Reference Guide as the FFS guide. In our investigation we do not attempt to provide a complete implementation for all of the API layers described in the FFS guide. We focus instead on a small subset of features that will allow creation and opening of file; as well as writing to, and reading from, files. To do this we restrict our attention to the File System Layer API, and a layer above this which represents the user of API, the application layer (see Figure 7.1). In our investigation the layers below the File System Layer are simulated, and in future work we could incorporate more detail of the lower layers based on the extension of our work presented in Chapter 8. The reason that we limit ourselves here to implementing just one layer of the API is due to the restrictions we place on OCB. The file system specification is hierarchical, each API layer may make calls into the API layer below it; in order to reflect this in our OCB specification instances defined in an API layer must

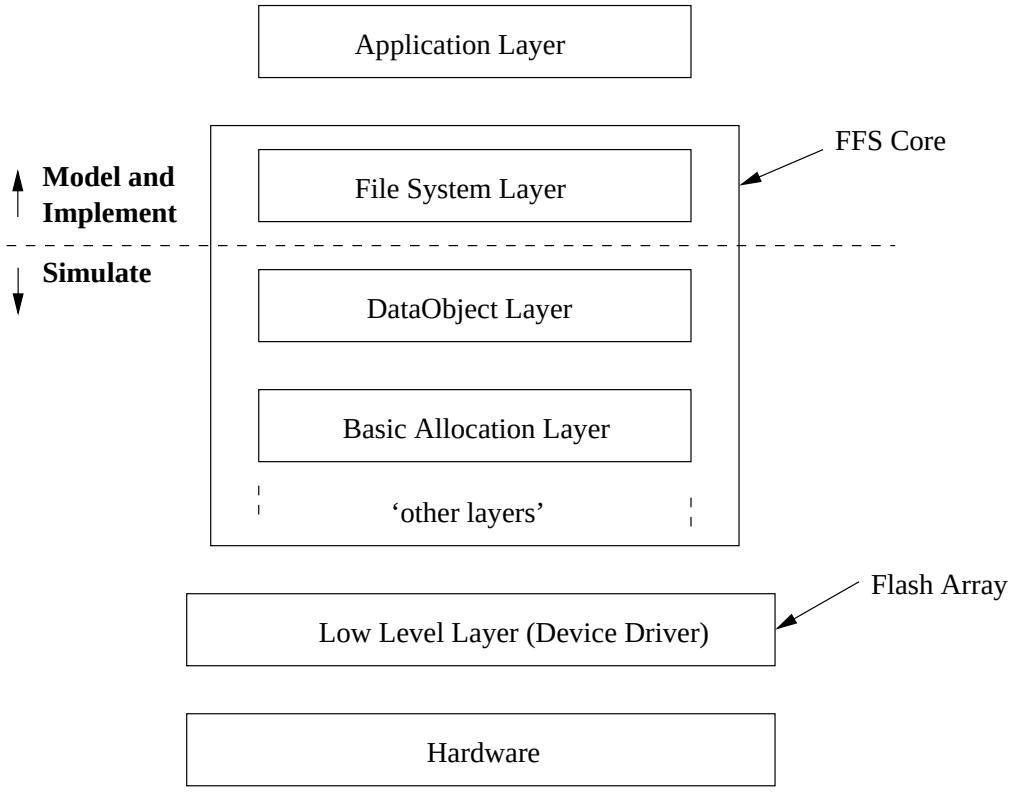


FIGURE 7.1: Flash File System Hierarchy

make calls to instances in the API layer beneath. In OCB the restriction that monitor objects cannot refer to other monitors (to prevent the nested monitor call problem and potential deadlocks) prevents easy access to lower API layers.

It is also worth noting at this point that the target implementation of the FFS guide is C [93]. In order to provide a Java implementation we create an Event-B abstraction of the implementation details that have been described using the C language, and then provide our Java implementation via translation of OCB specification.

7.1.1 The Flash File System API

The File System layer API specifies a number of features and we will look at just a few which are related to opening, reading from, and writing to, files. In the case study we model the capability to create and open a file, write to the file and read from the file. To do this we will first model in Event-B, and then OCB, the FFS API functions that we describe in this section. We are then able to translate the OCB to a Java implementation. In the description that follows we omit much of the detail in the FFS specification [82], since it is not relevant to our discussion here. For instance when we describe the functions of the FFS API we omit some of the parameters to aid clarity. The types used in the FFS guide include character strings, and types defined in the FFS guide itself. However, when modelling the system we use abstraction, and choose

to convert these more complex types, such as a files name, or access mode, to simple integer representations. These abstractions are also then carried into the simulated implementation. The C code used in the specification uses pointers, and we wish to abstract away from this also. Generally when we encounter a pointer indirection in the C specification we model the structure that the pointer refers to. As we move toward implementation using OCB, the pointer indirection in C relates to an attribute; and in Java, a field name.

We now show the first function header from the FFS specification, $FS\_WriteFile$, that we wish to implement.

$$\begin{aligned} FFS\_Status \;\; FS\_WriteFile(&FS\_FileHandle\ han,\\ &VOID\_PTR\ dat,\\ &UINT32\_PTR\ sze, \ldots) \end{aligned}$$

$FS\_WriteFile$ writes sze bytes from the buffer dat to the file specified by the handle han. Writing to han starts at the current offset, which is advanced as each byte is written.

Next we look at the function, $FS\_ReadFileDir$ which reads data from a file or directory.

$$\begin{aligned} FFS\_Status \;\; FS\_ReadFileDir(&VOID\_PTR\ buf,\\ &FS\_FileHandle\ han,\\ &UINT32\_PTR\ sze, \ldots) \end{aligned}$$

$FS\_ReadFileDir$ reads sze bytes from an open file identified by han into a buffer identified by buf, starting at the current file offset. The file offset is advanced as each byte is read.

In order to read or write to a file it must first have been created. The last function that we will present here is the $FS\_OpenFileDir$ function which we use to create and open a new file. The function may also be used to open an existing file, or create a directory, but we do not use it in this way here.

$$\begin{aligned} FFS\_Status \;\; FS\_OpenFileDir(&mOS\_char\ pth,\\ &FS\_FileHandlePtr\ han, \ldots,\\ &FS\_OpenMode\ omd,\\ &FS\_ShareMode\ smd, \ldots) \end{aligned}$$

The pth parameter is the full path (including file name) of the file to open. han is a pointer to the open file; omd is the open mode of the file, which dictates if a new file is always created, if an existing file can be overwritten to create a new file, and if it is to be open for read only, write only or read/write access. The smd parameter is used to specify whether the file is allowed to be shared, i.e. do not care about share, do not share

at all, only share among readers (fail on open for write), or only share among writers (fail on open for read). The return parameter of this function is of type FFS_Status which is used to report the success, or otherwise, of the request.

7.1.2 The Data Object Layer API

The File System Layer makes calls into the Data Object Layer using the Data Object Layer API. We have discussed the limitation of OCB to model nested objects, so we will be simulating this, and lower layers of the specification using an abstraction of the following API functions. *DO_AllocateDataObject* is called by the File System Layer to allocate space for data objects. This function call is used to allocate space for files, and directory objects, as well as data objects used for file system management. It is used when creating new data, and may be used to overwrite existing data.

$$\begin{aligned} FFS\_Status\ \ DO\_AllocateDataObject(&BA\_UnitLocationPtr\ dst,\\ &BA\_UnitLocationPtr\ src,\\ &BA\_UnitType\ type) \end{aligned}$$

The *dst* parameter is a pointer to the newly allocated data object. *src* is a pointer to an existing data object that is to be modified (or NULL if a new allocation is required), and *typ* is the type required. The type, of a new file/directory object is defined as $BA\_UnitTypeFileDir$ in the $BA\_UnitType$ definition.

To read a data object the Data Object Layer exposes the following function called *DO_ReadDataObject*. The *DO_ReadDataObject* function is typically called by functions of the layer immediately above, the File System Layer, in response to some request from the User Application Layer.

$$\begin{aligned} FFS\_Status\ \ DO\_ReadDataObject(&VOID\_PTR\ buf,\\ &BA\_UnitLocation\ obj,\\ &UINT32\_PTR\ sze,\\ &UINT32\ off, \ldots) \end{aligned}$$

The *DO_ReadDataObject* function transfers *sze* bytes of data from the data object identified by *obj* into the buffer identified by buf. The read begins at the offset specified by off.

$$\begin{aligned} FFS\_Status\ \ DO\_WriteDataObject(\ldots, &BA\_UnitLocationPtr\ obj,\\ &VOID\_PTR\ dat,\\ &UINT32\ sze,\\ &UINT32\ off, \ldots) \end{aligned}$$

$DO\_WriteDataObject$ is called by the File System Layer to write sze bytes from the buffer dat to the data object obj from the offset off.

7.2 Modelling the Flash File System

We present a model describing the behaviour at a high level of abstraction, and then a further refinement that we will implement. We introduce three events in our first model: *create* to perform file creation; *write* to perform writes to the flash memory; and *read* to perform reads from flash memory.

7.2.1 The Abstract Model

We begin by defining the context where we introduce some of the sets and constants of the development.

SETS

$DataObject\_Set$

CONSTANTS

$Data$

AXIOMS

$axm1 : Data \subseteq \mathbb{Z}$

The set $DataObject\_Set$ represents the potential data objects of the development. $Data$ represents the data that can be stored in data objects, we use integer data as a convenient abstraction; the integer representing an index into the dataset where the data resides. We now look at the variables of the abstract machine and their typing invariants,

VARIABLES

$DataObject,\ data,\ usr\_R\_Buff$

INVARIANT

$DataObject \in DataObject\_Set \wedge$

$data \in DataObject \pfun \mathbb{P}(Data) \wedge$

$usr\_R\_Buff \in DataObject \pfun \mathbb{P}(Data)$

$DataObject$ represents the set of instantiated data objects, $data$ represents the data associated with a particular data object. The locations that have been written form the

set *data*, we assume the range of locations from $\mathbb{P}(Data)$ are available. usr_R_Buff represents a buffer in the application layer and relates a data object to its data. The Initialisation event initializes the variables of the machine,

$$
\begin{aligned}
&INITIALISATION = \\
&\quad DataObject := \varnothing \| \\
&\quad data := \varnothing \| \\
&\quad usr\_R\_Buff := \varnothing
\end{aligned}
$$

The *create* event models the behaviour of the $FS\_OpenFileDir$ function with the parameter *omd* set to $FS\_CreateNew$, which is the flag indicating that a new file is required. The read and write events relate to, $FS\_ReadFileDir$, and $FS\_WriteFile$ respectively.

$$
\begin{aligned}
&create = \text{// create a new file} \\
&\quad \textbf{ANY} file \\
&\quad \textbf{WHERE} \\
&\quad\quad file \in DataObject\_Set \setminus DataObject \\
&\quad \textbf{THEN} \\
&\quad\quad DataObject := DataObject \cup \{file\} \| \\
&\quad\quad data(file) := \varnothing \\
&\quad \textbf{END}
\end{aligned}
$$

The *create* event models creation of a file data object. We model instantiation of a data object represented by the $file$ parameter, where the file data object is non-deterministically selected from the set of uninstantiated data objects, $DataObject\_Set \setminus DataObject$. We add the file to the set of $DataObject$ instances with $DataObject := DataObject \cup \{file\}$.

Next we look at modelling the *write* function,

$$
\begin{aligned}
&write = \text{// write a chunk of bytes from UserBuffer (to disk)} \\
&\quad \textbf{ANY } bytes, file \\
&\quad \textbf{WHERE} \\
&\quad\quad bytes \subseteq Data \wedge \\
&\quad\quad file \in DataObject \wedge \\
&\quad\quad file \in dom(data) \\
&\quad \textbf{THEN} \\
&\quad\quad data(file) := data(file) \cup bytes \\
&\quad \textbf{END}
\end{aligned}
$$

File System Layer	Abstract Events	Refined Events
FS_OpenFileDir	create	create
FS_WriteFile	write	w_start, w_step, w_end
FS_ReadFile	read	r_start, r_step, r_end

TABLE 7.1: File System Layer Abstraction and Refinement

The *write* event models writing a chunk of byte data represented by the parameter *bytes*, to a file data object, represented by the parameter $file$. The write is modelled using the action $data(file) := data(file) \cup bytes$ where bytes is added to the set of data of $file$.

We model the read behaviour in a similar way as follows,

$read =$ // read a chunk of bytes (from disk) to a buffer
ANY $file, bytes$
WHERE
$file \in DataObject \wedge$
$file \in dom(data)$
$bytes \subseteq data(file)$
THEN
$usr\_R\_Buff(file) := bytes$
END

The *read* event models the transfer of a non-deterministically selected chunk of a $file$'s data, represented by *bytes*, into the user's read buffer that is associated with $file$; the transfer is expressed by the action, $usr\_R\_Buff(file) := bytes$.

7.2.2 The Final Event-B Refinement

We now describe the last refinement step of the abstract development, just prior to the use of OCB to specify the implementation details. We introduce, to the model, processes initiated by the user in the User Application Layer that are used to write to, and read from, files. To model the set of user writing processes we add a set to the context, $UserAppWriteFile\_Set$. Table 7.1 shows the relationship between the File System Layer functions, the abstraction, and subsequent refinement, of interest to us in the case study.

As in the case study of Chapter 6 we use Jackson Structure type Diagrams, of [32, 86], to describe the refinement, and we remind the reader that the diagrams are an informal representation of the relationship between abstract events and events of refinements. They are used as an aid to understanding the correspondence between the events of

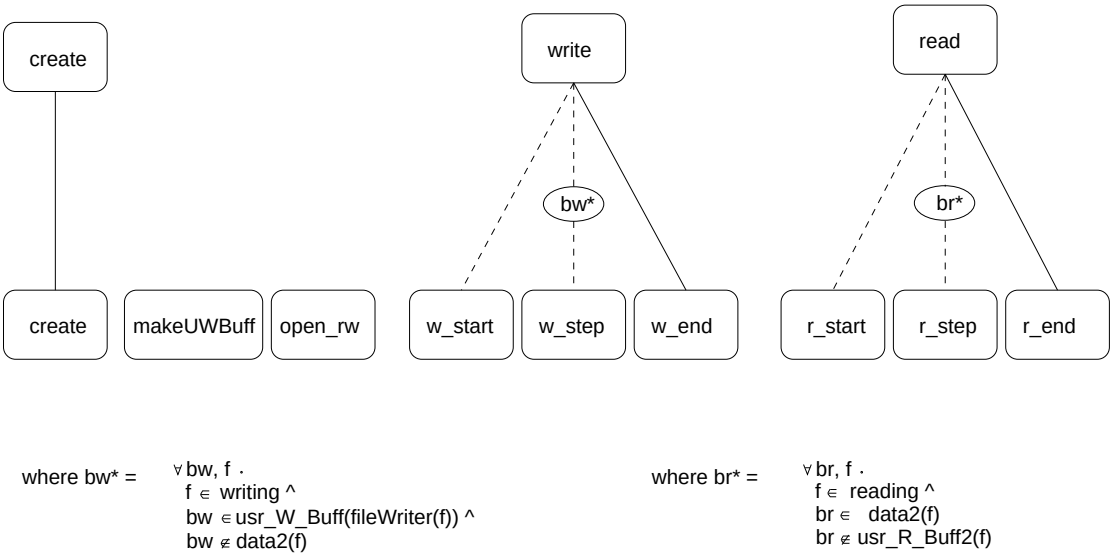


FIGURE 7.2: Abstract Development of the Flash File System

the abstraction and those of the refinements. Figure. 7.2 shows that the read and write steps have been split into a number of events with the solid line representing explicit event refinement. Iteration, where each byte is written from a user's buffer to a file, is represented by $bw*$; and where each byte is read from a file to a user's buffer is shown as $br*$. There are also additional events to create the user's write buffer, $makeUWBuff$; and an event which models opening a file for reading or writing, $open\_rw$. This corresponds to setting the file's access mode to $FS\_AccessReadWrite$.

When a file is to be used for reading or writing, it has to be opened first. It is at this point that the access permissions are checked, for instance, to see if a file has been previously opened by a user requiring exclusive access. It is at this point also that the current user requests exclusive access to the file. Due to this activity we do not consider the $open\_rw$ event to be part of the refinement of the $write$ event.

The $open\_rw$ event is as follows,

$open\_rw =$ // open file for read/write
REFINES $open\_rw$
ANY $file$
WHERE
$file \in DataObject$
THEN
$rwAccess := rwAccess \cup \{file\}$
END

$rwAccess$ represents a set of files that have been given read/write access on opening. The event describes a file data object that is non-deterministically selected and added to the set of files with read/write access using the action, $rwAccess := rwAccess \cup \{file\}$.

It can also be seen in Figure 7.2 that the write activity itself has been broken into three steps - w_start, w_step and w_end. w_end refines $write$ of the most abstract model. The next event we present is the first writing step w_start, which initiates the process of writing to a file by adding it to the set of writing files.

w_start = // initiate writing to disk(file.data)
from user write buffer

REFINES w_start

ANY $file$, $writer$

WHERE

$file \in rwAccess \wedge$

$file \notin writing \wedge$

$file \notin reading \wedge$

$writer \in UserAppWriteFile\_Set \wedge$

$writer \notin ran(fileWriter)$

THEN

$writing := writing \cup \{file\} \|$

$fileWriter(file) := writer$

END

The event guard $file \in rwAccess$ ensures that the file has read/write access, and $file \notin writing$ ensures that the file is not already writing where $writing$ represents the set of files that are writing. A similar guard exists to ensure the file is not reading. The $fileWriter$ is a function, $fileWriter \in DataObject \nrightarrow UserAppWriteFile\_Set$, which links a file instance to a writer process instance. In the next event we see that the writer contains a buffer with the data to be written, it has the type $usr\_W\_Buff \in UserAppWriteFile\_Set \nrightarrow \mathbb{P}(Data)$. The w_step event repeatedly copies a byte from

the user's write buffer to the file as follows,

$w\_step =$ // step writing bytes to disk
REFINES w_step
ANY $file$, $byte$
WHERE
$file \in writing \wedge$
$file \in dom(fileWriter) \wedge$
$fileWriter(file) \in dom(usr\_W\_Buff) \wedge$
$byte \in usr\_W\_Buff(fileWriter(file)) \wedge$
$file \in dom(data2) \wedge$
$byte \notin data2(file)$
THEN
$data2(file) := data2(file) \cup \{byte\}$
END

The file data is represented by $data2$, which relates files to data, $data2 \in DataObject \rightarrow \mathbb{P}(Data)$. w_step non-deterministically selects a byte in the user's buffer that has not already been copied. This is guarded by

$$byte \in usr\_W\_Buff(fileWriter(file)) \ \wedge \ byte \notin data2(file)$$

The other guards constrain the event so that only the file linked to the writing process write buffer can write to the file on the disk. The action,

$$data2(file) := data2(file) \cup \{byte\}$$

copies the byte from the buffer associated with the file $usr\_W\_Buff(fileWriter(file))$ to the disk represented by $data2(file)$. The file is removed from the set of file being

written when all the bytes have been transferred, in the $EndWrite$ event.

$endWrite =$ // finish writing from buffer
REFINES w_end
ANY $file$
WHERE
$file \in writing \wedge$
$file \in dom(data2) \wedge$
$\forall b(b \in usr\_W\_Buff(fileWriter(file)) \Rightarrow b \in data2(file))$
THEN
$writing \in writing \setminus \{file\}$
END

The file is guarded so that the $endWrite$ event is enabled when each byte in the user buffer is also on the disk, $\forall b(b \in usr\_W\_Buff(fileWriter(file)) \Rightarrow b \in data2(file))$. A similar approach was used to break the $read$ event of the abstraction into three events, where the r_end event refines $read$ but we do not show the specification here to avoid repetition.

7.2.3 An OCB Specification for Writers

After specifying an abstract development of the FFS we can provide implementation details which will allow translation to a Java implementation using OCB. Using our knowledge of the abstract development we identify a number of elements that can be implemented, for example we can see that some elements lend themselves to implementation as active process objects, such as readers and writers, that carry out tasks. We can also identify some elements that are shared between the process objects that do not play an active role, such as the data objects; we can implement these as shared monitor objects. Table 7.3 shows the main OCB classes that we introduce to implement the File System, for the full specification see Appendix D. The classes of the User Application

API Layer	Class	Type	Description
User Application	UserAppCreateFile	Process	User Invokes CreateFile
User Application	UserAppWriteFile	Process	User Invokes WriteFile
User Application	UserAppReadFile	Process	User Invokes ReadFile
User Application	UserBuffer	Monitor	Container for data
FFS	CreateFile	Process	Implements *create*
FFS	WriteFile	Process	Implements *write*
FFS	ReadFile	Process	Implements *read*
DataObject	DataObject	Monitor	Represents a data object

FIGURE 7.3: OCB Classes for the Flash File System

Layer specify processes which are created and run by users which, in turn, create files, and read or write to them by calling into the Flash File System Layer below it. We effectively have two processes associated with each create, read and write activity. This is a result of respecting the hierarchical nature of the development i.e. splitting into the User API and FFS API, and DataObject API; combined with the fact that procedure calls cannot be nested. We were forced to create another process to access the data in the *DataObject* API, which was referred to in an object of the FFS API. The $CreateFile$ process class implements the abstract *create* event, the $WriteFile$ process class implements the abstract *write* event, and the $ReadFile$ process class implements the abstract *read* event. The File System Layer processes call into the Data Object Layer below; but, since we are unable to make calls into the layers below the Data Object Layer, we simulate the activity of the Data Object Layer, where we allow a data object instance to store data in a buffer until it is full. To facilitate a realistic implementation we would have to accommodate acquisition of fresh data objects, to continue the write across a number of data objects when the current data object becomes full. The Data Object layer needs to invoke methods of the Basic Allocation Layer, directly below it in the API structure, to facilitate these activities. This layered structuring is not achievable in the current version of OCB due to the restriction on nested monitor classes.

Figure 7.4 shows the Jackson Structure type Diagram for the OCB implementation of the *write* event. The diagram is interpreted in the following way. Refinement of the abstract development is shown as in earlier diagrams. At the implementation level however, when no further refinement takes place, we interpret the diagram in a slightly different way. Sequences of OCB clauses identified by their labels are read from left to right. But branching gives rise to columns indicating a sequence within a branch. Iteration of a group of clauses is indicated by attaching the loop condition annotation to an enclosing box. It may be useful to make this notation more expressive in future, and we may wish to add additional features. For instance, in the case of clauses labelled $wf8\,..\,10$, the branches lead to process termination, and this could be defined on the diagram. A clause on the diagram may be associated with a false conditional branch, or false loop branch, which is not explicitly specified in textual OCB. This may occur, for instance, with **while** loop termination such as $wf12\_false$. We wish to include this on the diagram since $wf12\_false$ refines w_end. As a shortcut we group a sequence, $m\,..\,n$, of clauses with label name l, as $l_{m..n}$.

We now take a look at the OCB specification of the $WriteFile$ process class, which describes the process responsible for writing to a file. The process' behaviour is described in its *run* operation, shown in Figure 7.5. There is a close correspondence between the specification shown here and the diagram of Figure 7.4. The run operation's labelled clauses are shown in the diagram, as are the concepts of sequence, iteration and branching. Prior to creating a $WriteFile$ process the user application layer will have access to, or will have created, a store of open files - $OpenFileStore$ discussed in 7.2.4. The caller

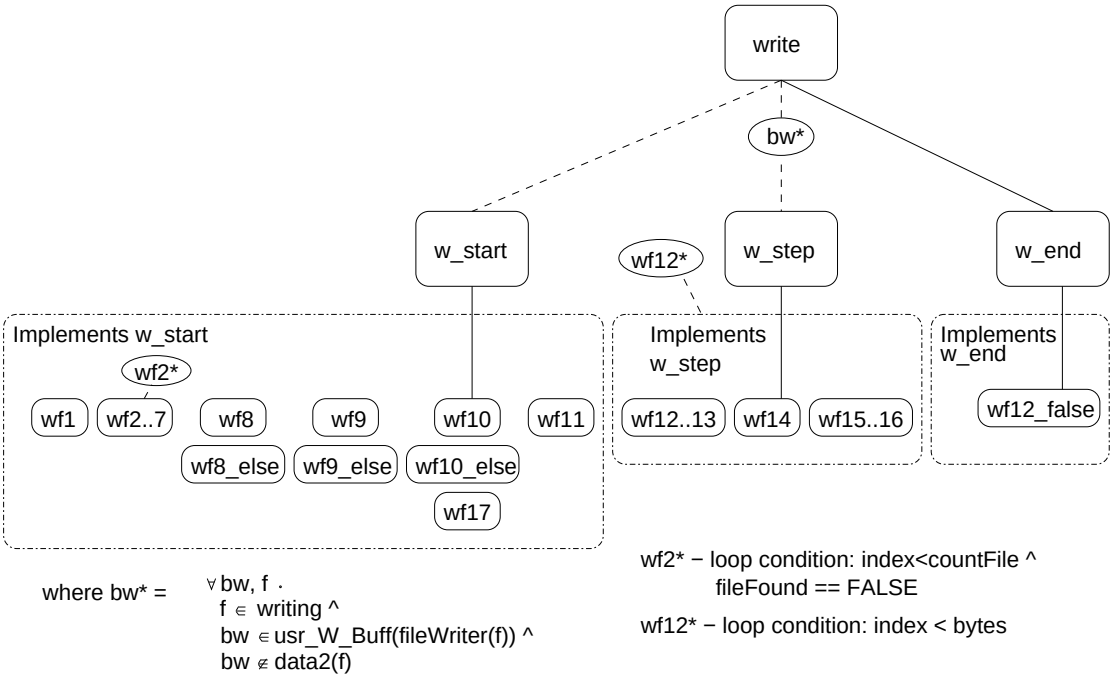


FIGURE 7.4: Diagram of the Refined FFS Write Event

of the $WriteFile$ class *create* procedure, in the application layer, provides the necessary parameters to allow the $WriteFile$ process to locate the file in the $OpenFileStore$ object.

The clauses labelled $wf2$ to $wf7$ perform a search of the store of open files, up to $openFileCnt$ which is obtained in clause $wf1$. If the file is found then the access mode is checked in $wf9$, otherwise an error is logged and the process terminates. The errors logged are; code 1 - id not found, code 4 - invalid access mode, and code 7 - data object full. The access modes are mode 0 - read only, mode 1 - write only, and mode 2 = read/write, and correspond with those defined in the FFS guide. If the access check fails then an error is logged in clause $wf9\_else$, otherwise the clause $wf9$ attempts to reserve space for the bytes to be written. Clause $wf10$ checks that this has been successful, and logs and terminates if not; otherwise all the checks have succeeded and we reset the file offset and write the bytes from the beginning of the file. It is intended that $wf14$ refines the w_step event; the event that models the write of a byte of data. We do not currently provide a refines clause in OCB to link the OCB specification and the abstraction, but it could easily be added in future work. In the meantime we add the refines clause manually as we also do for the witness clause.

```
ProcessClass WriteFile{
 // Attributes.
 OpenFileStore openFileStore, UserBuffer buffer, Integer id,
 Integer tmpName, OpenFileInfo file, Integer bytes, Integer index,
 Integer openFileCnt, Boolean fileFound, FileDirInfo fileDirInfo,
 Integer data, DataObject dataObject, Integer offset, Integer aMode,
 ErrorLog errorLog, Integer freeSpace

 // Constructor procedure.
 Procedure create(OpenFileStore openFileStor,Integer fName,
                  UserBuffer buffr,Integer byts,ErrorLog errorLg){
  openFileStore:=openFileStor||id:=fName||buffer:=buffr||bytes:=byts||
  index:=0||openFileCnt:=0||fileFound:=FALSE||tmpName:=-1||data:=-1||
  offset:=0||aMode:=-1||errorLog:=errorLg||freeSpace:=0
 }

 // Description of the process' behaviour.
 Operation run(){
  wf1: openFileCnt:=openFileStore.getSize();
  wf2: while(index<openFileCnt & fileFound=FALSE) do
   file:=openFileStore.getAtIndex(index) andthen
   wf3: dataObject:=file.getDataObject();
   wf4: fileDirInfo:=dataObject.getFileDirInfo();
   wf5: tmpName:=fileDirInfo.getID();
   wf6: if(tmpName=id) then fileFound:=TRUE endif ;
   wf7: index:=index+1 endwhile ;
  wf8: if(fileFound=TRUE) then aMode:=file.getAccessMode() andthen
   wf9: if(aMode=1 or aMode=2) then
    freeSpace:=dataObject.reserveSpace() andthen
    // Clause wf10 refines w_start
    wf10: if(freeSpace>0) then index:=0 andthen
     wf11: file.resetOffset();
     // Clause wf12_false, where ¬(index < bytes), refines w_end
     wf12: while(index<bytes) do data:=buffer.get(index) andthen
      wf13: offset:=file.getOffset();
      // Clause wf14 refines w_step
      wf14: dataObject.write(data, offset);
      wf15: index:=index+1;
      wf16: file.incOffset() endwhile  endif
    else dataObject.unReserve() andthen
     wf17: errorLog.add(7) endelse  endif
   else errorLog.add(4) endelse  endif
  else errorLog.add(1) endelse
 }
}
```

FIGURE 7.5: The WriteFile OCB Specification

7.2.4 MonitorClasses for the Flash File System Implementation

The repositories and structures of the FFS guide can be shared between processes and are implemented in OCB using the MonitorClass construct. We have defined the following monitor classes - *DataObject*, *DOStore*, $OpenFileInfo$, *OpenFileStore*, and $FileDirInfo$. See Appendix D for details. We continue with a brief overview of these classes, but note that we simulate the *DataObject* implementation since we are unable to nest calls to lower API levels. The *DOStore* is a repository of *DataObjects*. $OpenFileInfo$ is an implementation corresponding to $FS\_OpenFileInfo$, which stores information about the open files in the system including the access mode, current file offset and a reference to the *DataObject* itself. *OpenFileStore* is the repository in which the $OpenFileInfo$ objects are stored.

We now look at the specification of the *DataObject* monitor class in more detail. In the OCB model we specify the *DataObject* MonitorClass which provides an implementation corresponding to FFS data objects. Each *DataObject* has a *type* attribute. The value of the type attribute is an integer value defined in the FFS specification ($BA\_UnitTypeFileDir$ has integer value 128). The *DataObject* monitor class uses an integer array *data* to hold the data and has an attribute reference to $FileDirInfo$. $FileDirInfo$ is the implementation corresponding to the structure $FS\_FileDirInfo$ of the FFS guide. $FileDirInfo$ holds information about the file such as name (or its integer simulation in our case), create time, and file attributes. We need to retrieve the file name from this structure during execution of $WriteFile$'s *run* operation. The *DataObject* with its attributes and procedure headers can be seen in Figure 7.6.

Most of the procedures require no explanation here, but note that the file offset is not held in the data object. Incrementing the file offset, after a read or write, is the responsibility of the File System Layer. An $OpenFileInfo$ object, associated with the opened data object, keeps track of the file offset. In the File System Layer the $ReadFile$, or $WriteFile$, *run* operation invokes the $incOffset$ procedure to increment the value of the offset. At this point we should discuss an issue that is not yet fully resolved in our work. In the procedure body of $getFileDirInfo$ we have included a **when** clause to block the process when the $fileDirInfo$ attribute does not refer to an object. This is not a completely satisfactory solution, since we should either provide some error message or prevent the occurrence of this kind of error. To produce an error message would require a branching statement in the procedure (which is not currently part of OCB). The other alternative is to show that $fileDirInfo$ is not null when it is accessed. In Event-B we need to show that $DataObject\_fileDirInfo(WriteFile\_dataObject(self))$ is not undefined for a writing process, represented by the $self$ parameter. So we require that $WriteFile\_dataObject(self) \in dom(DataObject\_fileDirInfo)$ at the time of the call. We can do this by adding this property as an invariant clause in the machine and

```
MonitorClass DataObject{
 Integer type, FileDirInfo fileDirInfo,
 Integer[10] data, Integer freeSpace

 // The constructor procedure
 Procedure create(Integer typ,FileDirInfo fileDirInf){
  type:=typ || fileDirInfo:=fileDirInf ||
  freeSpace:=10
 }

 // Obtain this object's FileDirInfo object
 Procedure getFileDirInfo(){
   when(fileDirInfo /= null){
     return := fileDirInfo
   }
 }: FileDirInfo


 // Read the byte at the supplied offset
 Procedure read(Integer offset){... }: Integer

 // Write the supplied byte at the supplied offset
 Procedure write(Integer val,Integer offset){... }

 // Obtain the type of this data object
 Procedure getType(){... }: Integer

 // Reserve space for writing a byte
 Procedure reserveSpace(){... }: Integer

 // Unreserve space for reserved for writing
 Procedure unReserve(){... }
}
```

FIGURE 7.6: The DataObject OCB Specification

relating it to the program counter l of the caller as follows,

$$\forall s \in WriteFile \wedge WriteFile\_state(s) = l \Rightarrow$$
$$WriteFile\_dataObject(s) \in dom(DataObject\_fileDirInfo)$$

This gives rise to proof obligations which must be discharged to show the property holds, and we can also apply this approach to other OCB attributes, where it is necessary to prove non-nullity.

7.3 The Event-B Model of the OCB Specification

The OCB model of the Flash File System Layer contains many labelled clauses and we choose the clause labelled $wf14$ and show $WriteFile\_wf14$ as a typical event arising from the translation of a labelled clause. As a brief reminder, the clause $wf14$ is defined in the $WriteFile$ process class as follows,

```
wf14: dataObject.write(data, offset);
```

The *dataObject.write* procedure call will be expanded in line. The call is defined as follows,

```
Procedure write(Integer val,Integer offset){
   when(offset>=0 & offset < 10){
      data[offset]:=val}
  }
```

The *write* procedure takes val and $offset$ as parameters and will write the value at the appropriate index in the array. In this implementation the procedure is guarded to ensure the offset is in the bounds of the array. However, a more appropriate implementation is to guard this (in a conditional clause) in the caller and set an error flag if the condition is not satisfied. However the example is sufficient for our purposes. We now show the

result of the translation of clause $wf14$ to Event-B.

$WriteFile\_wf14 =$

REFINES w_step

ANY $self,\ target$

WHERE

$self \in WriteFile \wedge$

$self \in dom(WriteFile\_state) \wedge$

$WriteFile\_state(self) = wf14 \wedge$

$self \in dom(WriteFile\_dataObject) \wedge$

$target = WriteFile\_dataObject(self) \wedge$

$WriteFile\_offset(self) = 0 \wedge$

$WriteFile\_offset(self) < 10$

WITH

$file = WriteFile\_dataObject(self) \ \wedge \ byte = WriteFile\_data(self)$

THEN

$DataObject\_data(target) :=$

$DataObject\_data(target) \Leftarrow$

$\{WriteFile\_offset(self) \mapsto WriteFile\_data(self)\} \parallel$

$WriteFile\_state(self) := wf15$

END

The translated event refines the w_step event since its action is to write the byte to the file data object. In the final refinement of the abstract development the byte is added to the data object $data2$ as follows,

$$data2(file) := data2(file) \cup \{byte\}$$

In the implementation model we relate the abstraction to the implementation model using two witnesses. The first witness represents the file being written to, $file = WriteFile\_dataObject(self)$, and therefore from the guard $file = target$. The second witness represents the byte being written to the file, $byte = WriteFile\_data(self)$. In the implementation the data to write is held in the buffer of the $WriteFile$ instance, represented by $WriteFile\_data(self)$.

$$DataObject\_data(target) :=$$
$$DataObject\_data(target) \Leftarrow$$
$$\{WriteFile\_offset(self) \mapsto WriteFile\_data(self)\}$$

Additionally we have $self$ which parametrises the writing process where $self \in WriteFile$, and an offset value $WriteFile\_offset(self)$ for the data.

We add the following invariant relating the written data of data object f, $data2(f)$ of the abstraction, with $ran(DataObject\_data(WriteFile\_dataObject(s)))$ the written data of the implementation. We are interested in ensuring that the written data is the same in the abstraction, and in the implementation refinement after writing is complete. Writing is complete when $WriteFile\_state(s) = wf14 \wedge \neg(WriteFile\_index(s) < WriteFile\_bytes(s))$, and is described in the events w_end and the refined event, $WriteFile\_while\_wf12\_false$.

$$\begin{aligned}
&\forall f, s \cdot\ f \in DataObject\ \wedge\ s \in WriteFile\ \wedge \\
&WriteFile_state(s) = wf12\ \wedge \\
&\neg(WriteFile_index(s) < WriteFile_bytes(s)) \\
&\Rightarrow \\
&data2(f) = ran(DataObject_data(WriteFile_dataObject(s)))
\end{aligned}$$

During the writing steps when there are bytes left to write, $WriteFile\_index(s) < WriteFile\_bytes(s)$, we equate the data, $data2(f)$, of the abstraction, with $ran(DataObject\_data(WriteFile\_dataObject(s)))$ in the following invariant. This ensures that the implementation writes satisfy the writes specified in the abstraction, the write steps are described in the event w_step and the refined event, $WriteFile\_wf14$.

$$\begin{aligned}
&\forall f, s \cdot\ f \in DataObject\ \wedge\ s \in WriteFile\ \wedge \\
&WriteFile_state(s) = wf14 \\
&\Rightarrow \\
&data2(f) = ran(DataObject_data(WriteFile_dataObject(s)))
\end{aligned}$$

Similar invariants ensure the data reads of the implementation satisfy the abstraction.

At run time the program can accommodate the fact that a call may be made to an object that does not exist. We provide a default handler for a monitor procedure call where the target is a null reference - the process simply terminates. In the implementation this corresponds to throwing an exception and terminating. We can optionally specify some activity performed before the process terminates, such as logging the error. Monitor class procedures may also contain null references, these are not currently handled satisfactorily; the process simply blocks. A mechanism could be introduced in future

work to ensure non-nullity as discussed previously in 7.2.4

$$
\begin{aligned}
&WriteFile\_wf14\_isNull = \\
&\quad \textbf{ANY } self \\
&\quad \textbf{WHERE} \\
&\qquad self \in WriteFile \wedge \\
&\qquad self \in dom(WriteFile\_state) \wedge \\
&\qquad WriteFile\_state(self) = wf14 \wedge \\
&\qquad \neg(self \in dom(WriteFile\_dataObject)) \\
&\quad \textbf{THEN} \\
&\qquad WriteFile\_state(self) := terminatedWriteFile \\
&\quad \textbf{END}
\end{aligned}
$$

In this section we have shown the events arising from the translation of the labelled OCB clause $wf14$ which refines the *w_step* event. The translation of other process clauses give rise to events modelling creation of files, and reading from, and writing to them. The top layer OCB specification consists of a $MainClass$ which is the entry point for execution. The user's Application Layer operations are called from the $MainClass$. The File System Layer operations, described above, are then called from within the user's Application Layer.

7.4 The Java Implementation

We now discuss the Java code that arises from the translation from OCB. We will continue with our discussion of the writing process with the $WriteFile$ Java class shown in Figure 7.7, and the *DataObject* class as an example of a monitor class, shown in Figure 7.8. The $WriteFile$ process class defined in OCB gives rise to the $WriteFile$ Java class that implements java's *Runnable* interface for defining the thread behaviour. The OCB attributes of the $WriteFile$ process class are translated to private Java fields which are initialized in the constructor method. The *run* method consists of the translations arising from the process class's *run* clause, with its labelled atomic clauses; the Java code closely resembles the OCB specification in this section. The Java code for the *DataObject* class closely resembles the OCB specification, and we have not reproduced it in its entirety. We just point out that OCB attributes map to private Java Fields, and that synchronized methods are used to implement procedures.

```
public class WriteFile implements Runnable{
 private OpenFileStore openFileStore = null;
 private UserBuffer buffer = null; private int id;
 private int tmpName; private OpenFileInfo file = null;
 private int bytes; private int index; private int openFileCnt;
 private boolean fileFound; private FileDirInfo fileDirInfo = null;
 private int data; private DataObject dataObject = null;
 private int offset; private int aMode;
 private ErrorLog errorLog = null; private int freeSpace;

 public WriteFile(OpenFileStore openFileStor, int fName,
       UserBuffer buffr, id = fName; int byts, ErrorLog errorLg){
  openFileStore = openFileStor; buffer = buffr; bytes = byts;
  index = 0; openFileCnt = 0; fileFound = false; tmpName = -1;
  data = -1; offset = 0; aMode = -1; errorLog = errorLg;
  freeSpace = 0;}

 public void run(){
  openFileCnt = openFileStore.getSize(); // wf1
  while(index < openFileCnt && fileFound == false){
        file = openFileStore.getAtIndex(index); // wf2
        dataObject = file.getDataObject(); // wf3
        fileDirInfo = dataObject.getFileDirInfo(); // wf4
        tmpName = fileDirInfo.getID(); // wf5
        if (tmpName == id) {
            fileFound = true; /*wf6*/ }
        index = index + 1; /*wf7*/ }
  if(fileFound == true){
        aMode = file.getAccessMode(); // wf8
        if(aMode == 1 || aMode == 2) {
            freeSpace = dataObject.reserveSpace(); // wf9
            if(freeSpace > 0){
                index = 0; // wf10
                file.resetOffset(); // wf11
                while(index < bytes){
                    data = buffer.get(index); / wf12
                    offset = file.getOffset(); // wf13
                    dataObject.write(data, offset); // wf14
                    index = index + 1; // wf15
                    file.incOffset(); /*wf16*/}}}
            else{ dataObject.unReserve();
                  errorLog.add(7); /*wf17*/}}
        else{errorLog.add(4);}}}
  else{errorLog.add(1);}}
}
```

FIGURE 7.7: The WriteFile Java Code

```
public class DataObject {

 private int type; private FileDirInfo fileDirInfo = null;
 private int[] data = new int[10]; private int freeSpace;

 public DataObject(int typ, FileDirInfo fileDirInf){
   type = typ; fileDirInfo = fileDirInf; freeSpace = 10; }

 public synchronized FileDirInfo getFileDirInfo(){... }

 public synchronized int read(int offset){... }

 public synchronized void write(int val, int offset){...}

 public synchronized int getType(){... }

 public synchronized int reserveSpace(){... }

 public synchronized void unReserve(){...}
}
```

FIGURE 7.8: The DataObject Java Code

7.5 Issues Arising from the Case Study

The motivation for the case study was to attempt to provide an implementation of part of the flash file system specified in the Intel Flash File System Core Reference Guide (FFS Guide) [82]. In particular we wanted to see how an OCB implementation could be used to implement part of an existing system, and the problems that would be encountered. In related work, modelling of the flash memory has been undertaken using Z [36] but deals with lower level parts of the ONFI specification [83]; our interest in this case study is specification at higher level. VDM++ has also been used to create a model [58] based on the FFS guide, the paper presents the activity of deleting a file from the file system. The resulting model was translated to, and analysed with, Alloy [85] and HOL [3]. A key feature of the VDM++ work was the integration of the tools and techniques, our main focus is the integration of the Event-B method and an object-oriented implementation that makes use of concurrency. The Event-B approach and Rodin toolset have been designed, from the outset, to simplify the process of specifying and discharging proofs. In some respects the expressiveness of the tools have been limited, in order to support a simpler approach to proof. In other respects Event-B can be seen as much more general since it is not limited to modelling software systems. Approaches such as Z and classical-B support a design by contract approach using preconditions and postconditions and are therefore much more tailored to modelling software systems. In this way one could view Z and classical-B as more expressive than Event-B in that they provide more language structures; but the simplicity of Event-B can be seen as providing a

much clearer view of a system as development proceeds. The Rodin tool and Event-B approach were developed to be used together which gives rise to an integrated approach. This approach of VDM++ work above differs since the VDM++ model of the file system was translated to Alloy and HOL. In the Z model of flash memory the proof was not discussed, but typically the Eves prover [129] would be used since it has been tailored for use with Z.

The FFS Guide describes a complex system and a number of APIs, layered in a hierarchical fashion, are used to partition the specification. It quickly became apparent that our OCB implementation would not be able to follow the hierarchical structure, and attention was restricted to an implementation of the User's Application Layer, and the File System API only. With the Data Object layer and those below being simulated. The problem occurs mainly because we prevent monitor class procedures from invoking procedure calls of other monitor classes, a restriction imposed in order to prevent deadlock and the nested monitor problem. This demonstrates the need for an extension to existing work that has a more flexible approach. In an implementation that does not block the process on a failed lock acquisition attempt, we would be able to accumulate a number of object locks without fear of deadlock. If we encountered a lock that was held by some other object we could simply release the accumulated locks. If we are able to accumulate locks then we may then be able to entertain the notion of nested monitor objects and allow procedure calls on them. In the next chapter we discuss our extension of the OCB approach which should allow a transactional approach, which will allow us to access multiple objects from within a labelled clause. However, the OCB notation seemed to be most suitable for the specification of the User Application Layer and the Flash File System Layer, which is where the processes of the system are invoked.

We initially described the part of development we are interested in using an abstract model where readers and writers copy a chunk of bytes to disk atomically. We then refined the model to model the transfer of the bytes individually, that make up a chunk of data. We then presented a systematic approach to linking the refined model with an OCB specification which resulted in a translation to an Event-B model and Java code. We found that the Jackson Structure type Diagrams were a useful aid to visualising the relationships between the abstract events and the events of the refinement at the implementation level. The Jackson Structure type Diagrams can embody sequencing, iteration and branching; and we see how this links to sequence, looping, and branching in the OCB specification.

In the OCB approach a single monitor procedure call can be invoked in a labelled atomic clause. However, it is not always possible to know in advance when a target refers to an object or is undefined. There are approaches that attempt to ensure this within certain constraints, such as Spec# [23]; but this is not a general solution since it does not cater for non-null types in arrays. We have handled this approach by modelling process termination, where in the Java implementation the thread throws an exception

and terminates. We showed an additional associated problem, where an attribute inside a procedure body may be undefined. We showed that we could add an invariant and, in this case, use proof to show non-nullity, and we think that this approach could also be used to prove non-nullity in most other circumstances.

The OCB specification consists of a $MainClass$, 6 process classes, and 7 monitor classes. There are 62 labelled clauses in total. These give rise to 127 events and 113 typing invariants in the Event-B model. The number of proof obligations generated from just these (before proving refinement) totalled 827, 777 of which were discharged immediately by the auto-prover. A further 77 were discharged relatively easily and 18 remain to be discharged. The size of the translated Event-B model caused some problems, with the automatic prover taking a very long period of time to run through the proof obligations. The problem that this caused indicates that we should consider some form of modularity, or decomposition of the model into more tractable partitions. The proof of refinement has not yet been completed, but there is room for including some productivity enhancements such as adding witnesses and refines clauses to OCB operations, thus indicating their relationship with the abstract events. The translation from the OCB specification to Java code worked as expected, and we were able to run the program in the JVM.

The user interface, in its current state, is suitable only to for use in investigating, and experimenting with, the approach. We make use of the Eclipse tree editor which is not well tailored to our needs. It would seem natural to use a UML-like diagram editor with class diagrams, such as that used in the UML-B tool, to specify our process and monitor classes. This would then provide a natural progression from Event-B modelling, using UML-B, to OCB specification.

Chapter 8

Extending OCB with Transactional Constructs

In this chapter we propose a transactional version of OCB constructs which makes use of a later Java platform version that has greater control over locking and conditional waiting. The transactional version of OCB introduces a number of features which overcome the limitations of the OCB described in previous chapters. We first discuss the newer features of Java, introduce the new syntax, and then describe mappings to Event-B and Java.

8.1 The Java Language Specification - Third Edition

There were many concerns about the Java Language Specification - Second Edition (JLS 2)[37], which was shown to contain many ambiguities and omissions. Problems most typically manifested themselves in systems where concurrent execution of threads was employed; to the extent that sharing of data that could modified was eliminated by design if at all possible, as suggested in [100]. Much has been written on the subject of overcoming concurrency problems in Java, some by proposing extensions to the language, or adding annotations to Java [22, 25, 62, 81, 88, 92], or some by suggesting how to work with this, and the previous, Java edition with careful checking [16, 61, 80].

The memory model has been redefined in the Java Language Specification - third edition (JLS 3)[68] which serves to define the language up to version 1.5 of the Java SDK. The specification has a more detailed description of the read and write actions of threads to overcome the ambiguities in the previous version; and there are new features to assist with programming for concurrency. There is an atomic Compare and Set (CAS) operation associated with objects that have atomic updates that makes use of atomic

primitives recently introduced in the latest processor technology. It has the form,

boolean *compareAndSet*(*expected*, *newVal*)

where the *newVal* is assigned to the object, and returns **true**, if the value of the object equals *expected*; else returns **false** and makes no change. JLS3 makes use of this new feature to provide the *AtomicInteger*, *AtomicBoolean* and *AtomicReference* types, among others, which are part of the *java.util.concurrent.atomic* API. The *java.util.concurrent* API provides a Semaphore class which allows threads to acquire permits, and block if no permit is available. A semaphore can make a number of permits available in order that each object of a pool of resources can be acquired; each acquisition is atomic. Permits are released when no longer required by a thread and other threads are notified and may compete for the resource. A semaphore with a single permit can function as a mutex lock. The semaphore makes use of the new atomic constructs such as *AtomicInteger* and *compareAndSet* in its implementation.

Another feature of JLS3, that is an improvement over JLS2, is in the area of locking objects and conditional waiting. JLS3 introduces the *java.util.concurrent.locks* API which provides features that allow explicit control over object locking. It is intended that the new locking constructs should not be used in conjunction with the synchronization mechanism of JLS2 since the two mechanisms are independent of each other. One useful feature that we can make use of is non-blocking lock acquisition. In JLS2 if an object's monitor lock is held by a thread, if a second thread attempts to acquire the lock then it is blocked. The developer has no control over this behaviour, JLS3 enables the creation of *Lock* objects with a *tryLock* method. The *tryLock* method will simply return **false** if the lock is already held by some other thread, thus facilitating non-blocking lock acquisition. A lock can also be released using the *unlock* method. The API also provides a *ReadLock* that can be shared among readers and a *WriteLock* that allows mutually exclusive access, but waits until all readers have finished reading.

Conditional waiting has been enhanced by the use of explicit *Condition* objects upon which a thread may wait. A new Condition object is obtained from an existing *Lock* object by the use of the *newCondition* method. The *Condition* object's *await*, *signal* and *signalAll* methods perform the same function as the *wait*, *notify* and *notifyAll* methods. The advantage of explicit *Condition* objects is that a number of such objects can be associated with a single lock. Each can have its own condition attached, and is therefore more flexible than the old method. Of the Lock types that are specific to read and write activities, it is possible to associate a *Condition* object with a *WriteLock*, but not a *ReadLock* - an exception is thrown if this is attempted.

8.2 Transactional Constructs

We are able to use the new language features of JLS3 to implement a more flexible OCB approach, and introduce access to a number of objects in an atomic clause - rather than restrict access to one object as is the case of out initial OCB approach. Since we are able to access more than one object in a labelled atomic clause we refer to the extension of our approach as Transactional-OCB. When we compare the original approach we will refer to it as Synchronized-OCB, due to its reliance on this method of locking objects. Figure 8.1 is an aid to visualising the difference between monitor locks and the use of a lock manager. The reader should note at this point are discussion of object locking is an implementation issue - the locking scheme should remain largely transparent to the user of OCB. The only exception may be any restrictions that we introduce to ensure that the approach is transactional. So, with respect to the implementation of Synchronized-OCB, a process can only obtain a single lock for an object; this is facilitated by the use of Java's synchronized method calls. In contrast, the lock manager of the Transactional-OCB approach is able to gather a number of object's locks; and furthermore, it is able to release the held locks if the manager is unable to acquire any particular lock of a set of required locks. In Transactional-OCB each labelled clause is associated with a set of locks that it requires; and the set of locks is to be acquired before entry into the critical region of the clause. It is for this reason that all the locks should be known prior to the clause body being executed. If we cannot decide which locks are required before entry to the critical region then it may be the case that one of those is not available and the transaction could not proceed. This would be problematic in the middle of a transaction since we have no roll-back facility, any state updates that we have made would be permanent. To ensure that procedure calls are decidable in advance we impose the restriction that if multiple procedure calls are made in a labelled atomic clause then the procedures that are called are getter methods only, that is, when determining which locks are required to effect the procedure call, they will not update state before entry to the critical region. This restriction can be checked statically prior to the automatic translation begins. So the calls may be nested, but they must in turn only be to getter methods. In this way we avoid having computations involving state updates at the lock acquisition stage.

We now present some examples of Transactional-OCB clauses in which we make use of two procedure definitions, *add* and *sub* specified as follows,

$$\textbf{Procedure}\ add(Integer\ a)\{\ x := x + a\ \}$$

The *sub* procedure subtracts an amount from x and returns the new balance,

$$\textbf{Procedure}\ sub(Integer\ a)\{\ x := x - a\ ;\ \textbf{return} := x\ \}\ :\ Integer$$

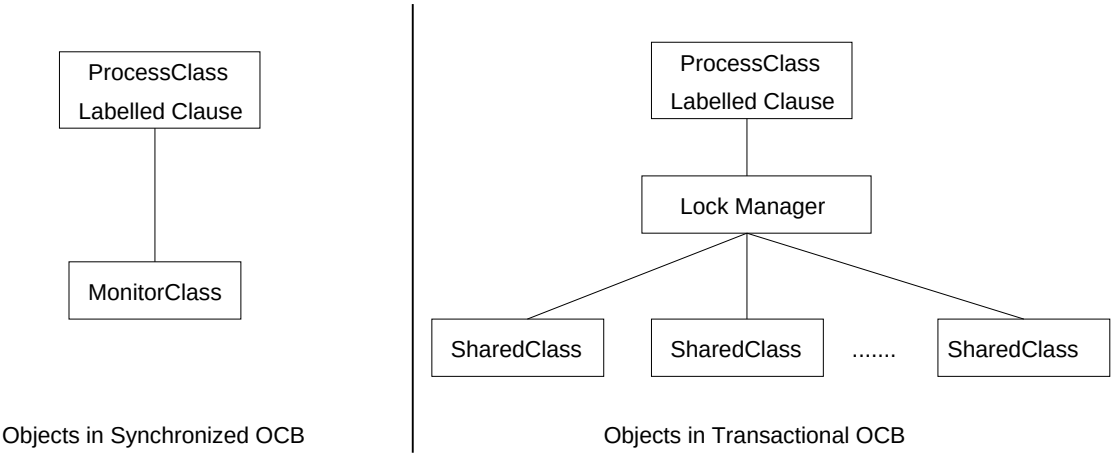


FIGURE 8.1: OCB Locking Strategy

The following labelled atomic clause involves a sequence of updates to different objects using the procedure calls defined above, note that we now allow more than one procedure call in a labelled clause.

$$l1: \ \triangleleft \ m1.sub(amt) \ ; \ m2.add(amt) \ \triangleright$$

The brackets enclosing atomic regions, $\triangleleft$ and $\triangleright$, are introduced to the specification here. We use these to make it clear which areas are contained within an atomic region. The sequence operator, used in labelled atomic clauses, does not permit interleaving so in this sense the operator differs from the sequence operator used in non-atomic clauses. It is simply used to sequence updates within an atomic region. The sequence operator makes the use of the parallel operator redundant in the actions of labelled atomic clauses. In the preceding clause the object referred to by attribute $m1$ has its *sub* procedure called and is passed the parameter *amt*, then the object referred to by $m2$ has its *add* procedure invoked with the parameter *amt* being passed.

Using the above clause we can atomically update two distinct objects without interference by some other process, or threat of deadlock. The type of deadlock situation that may arise here is caused by resource contention conflicts. We discussed interference and deadlocks earlier in Section 2.8.2. Deadlock prevention is achieved using explicit lock objects, and the *tryLock* method, where an unsuccessful lock acquisition attempt releases any held locks. We are also able to accommodate OCB **when** clauses in these objects since we can wait on a condition after releasing the locks already held.

We also introduce direct access to owned object attributes which can be seen in the following clause. In Synchronized-OCB a labelled clause may only update the attributes of a shared object through a procedure call, in Transactional-OCB we can make use of direct access to objects referred to by attributes. This can however be viewed as

syntactic sugar; an alternative to the use of procedure calls.

$$l1: \ \triangleleft \ t2 := m2.x \ ; \ m2.x := m1.x \ ; \ m1.x := t2 \ \triangleright$$

We are now able to make the actions of labelled atomic clauses more expressive, and would like to add some other features that correspond to implementation level constructs. We introduce a branching clause for use in our labelled atomic clauses which is used as follows,

$$\begin{aligned} l1 : \triangleleft \ & \textbf{if}(m1.getx() - amt \geq 0) \ \textbf{then} \\ & m1.sub(amt) \ ; \ m2.add(amt) \ \textbf{endif} \ \triangleright \end{aligned}$$

In this clause the whole branch is enclosed in atomic brackets. This branching clause differs from the non-atomic version since the non-atomic version allows interleaving between the atomic part and the '**andthen**' clause. The branching construct works in the usual way with a conditional part, and an action that is evaluated if the condition is true. The clause may also have a number of **else** branches to describe alternative branches. We also extend the notation to allow invocation of a procedure call in the condition, providing the call just retrieves a data value from an object and is free of side-effects. We will discuss restrictions on procedure invocation in detail when we discuss lock management later. In this branching clause, then, conditional evaluation and any subsequent updates occur atomically. The branching clause may also co-exist with other clauses which will also form part of the same atomic update.

The following atomic **while** construct will also be useful for performing loops in labelled atomic clauses. It once again differs from the non-atomic looping construct introduced earlier, since it does not provide a mechanism for interleaving at the end of each loop iteration.

$$\begin{aligned} l1 : \ \triangleleft \ & \textbf{while}(i > 3) \ \textbf{do} \\ & t1 := m1.getx() - 1 \ ; \ t2 := m2.getx() + 1 \ ; \\ & m1.setx(t1) \ ; \ m2.setx(t2) \ ; \ i := i - 1 \ \textbf{endwhile} \ \triangleright \end{aligned}$$

The looping construct consists of a condition which is evaluated and an action that is performed if the condition is true, else the loop terminates if false. It may be desirable to prove that a loop can always reach its termination condition, in which case a natural number loop variant can be specified. The clause's action must always decrease the variant; in the example above a suitable loop variant is $i - 3$, which is decreased by the action $i := i - 1$.

8.2.1 Transactional-OCB syntax

In our approach using transactional constructs we re-use the non-atomic syntax of earlier chapters. In the Java implementation we can protect critical regions using either a lock-based management scheme, for which we have a prototype; or we could implement a solution based on Software Transactional Memory (STM) as discussed in [73, 74, 75]. STM provides a transaction-based solution that maps well to the transactional constructs of OCB. STM constructs provide the usual commit and abort utilities. One drawback with an STM implementation is the overhead associated with keeping track of the changes data; all changes by transactions are recorded temporarily, regardless of whether a transaction succeeds in committing. This may be an inefficient use of memory space in a system where contention is high. There is also a time overhead associated with committing resources. A more memory efficient implementation may prevent changes to the data being made by locking the data objects, reducing the amount of modified data in memory, but increasing the time processes spend waiting (a reduction of the amount of concurrent processing being done). We give more details of the STM approach and our lock-based implementation later in the chapter. We choose a lock-based implementation over an STM style implementation for our work for simplicity. In this extension to our work we retain the concepts of process classes and introduce the notion of shared classes which we use instead of monitor classes. Re-capping, the syntax for a process class is defined as,

$$ProcessClass ::= CName\ Var^{+}\ NonAtomic\ Constructor$$

and for shared classes we have,

$$SharedClass ::= CName\ Var^{*}\ Procedure^{+}\ Constructor$$

Process classes have a name $CName$, a set of instance variables Var, a non-atomic clause declaration $NonAtomic$, and a constructor procedure $Constructor$. Shared classes are similar to process classes but have a set of procedures $Procedure$ instead of a non-atomic clause. The definition of $NonAtomic$ remains the same.

$$\begin{aligned} NonAtomic ::= & \\ & NonAtomic\ ;\ NonAtomic \\ & |\ NonAtomic\ []\ NonAtomic \\ & |\ do\ Atomic\ [;\ NonAtomic]\ od \\ & |\ Atomic \end{aligned}$$

We modify the previous definition of a labelled atomic clause, replacing *Body* with *Action*, since *Action* now contains all the clauses we require,

$$Atomic ::= StartLabel : \ \triangleleft \ [Guard \rightarrow] \ Action \ \triangleright$$

The guard that follows the start label is related to the wait condition *when* construct. In Transactional-OCB we move conditional waiting out of procedure definitions - up to the level of the *Atomic* clause. An example of the textual definition follows where a labelled clause with a **when** construct contains an action,

$$l1 : \ \triangleleft \ \mathbf{when}(c)\{a\} \ \triangleright$$

where $c \in Guard$ and $a \in Action$

If condition c is false then the process will block. In the implementation the thread is blocked by invoking the await method, and waits for a signal from some other thread, we will describe this more fully later in the chapter. Another difference between synchronized and Transactional-OCB is that we allow procedure calls in the conditions of looping and branching constructs, but only where the procedure is free of side-effects, and returns an appropriate value (which can be checked statically). We require freedom from side-effects since a condition should contain only a predicate guard.

An *Action* in Transactional-OCB differs from Synchronized-OCB since actions can be composed using a sequence operator, the parallel operator is no longer used in Transactional-OCB. This introduces the requirement for a sequence operator in the Event-B language which does not exist at the time of writing; we do not discuss the ramifications of introducing this to Event-B, at this stage, and simply assume that it is possible to do so. We know that the sequence operator existed in classical-B below the abstract machine level; in refinements and implementation machines. We therefore have some confidence that it is feasible, and we know that one of the reasons that it was not included in Event-B was to reduce complexity. We argue that it will be useful (and indeed necessary) to be able to describe this kind of sequential behaviour when working at the implementation level. We now discuss the justification for not having the sequence operator in Event-B, presented in [72]. We know that an invariant I involving variables x and y can be written as $I(x, y)$. We wish the invariant to hold for the following program,

$$x := E(x, y) \ ; \ y := F(x, y)$$

To show that the invariant holds we have the following proof obligation which we must discharge,

$$I(x, y)$$
$$\vdash$$
$$I(E(x, y), F(E(x, y), y))$$

We can see that expressions on the right-hand side of assignments are carried forward, and become nested in the proof of subsequent assignments. The complexity does increase as the number of clauses in the composition increase. However, in our work at the implementation level, we will not use non-deterministic constructs. This therefore reduces some of the complexity alluded to in the justification for not including the sequence operator. We also introduce branching and looping constructs to actions without in-depth discussion; this will of course involve the introduction of the corresponding constructs in Event-B. Branching constructs were included at all levels of classical-B modelling, from abstract machines down; and looping was included at the implementation level. The assignment clause used in Transactional-OCB is the same as that of the Synchronized-OCB syntax. Another difference from the previous *Action* syntax is the ability to make a number of procedure calls in an action - in Synchronized-OCB we were restricted to just one call per labelled atomic clause; in addition we now allow direct access to another class's attributes whereas in Synchronized-OCB all such accesses were via procedure calls. This leads us to redefine our syntax where once we used a simple attribute v in our definition, we now expand the syntax to include the notion of an attribute v which can either be a name; a composition of attribute names using the dot-notation; or an array access. The syntax for v follows,

$$\begin{aligned} v ::= & \\ & identifier \\ & \mid identifier \text{`['} IntegerLiteral \text{`]'} \\ & \mid identifier \text{` . '} identifier \end{aligned}$$

Procedure calls, of Synchronized-OCB's *Body* clause, are subject to some rigid constraints. Procedures are only allowed to be called by process classes and the target must be a shared class. In Transactional-OCB we define procedure calls as part of an *Action* and relax some of the constraints. We allow calls to locally defined procedures (calling of procedures defined within the same process or shared class as the caller), shared class procedures (as before but multiple calls in an action), and constructors. We still, however, wish to prevent recursive procedure calls - Event-B proof rules would need to be developed to handle recursion. So we need to ensure the absence of recursion statically, or prohibit calls to procedures that have calls themselves. In Transactional-OCB we do not permit the use of the *when* clause (for conditional waiting) in procedures. Since

an atomic clause can now contain a number of procedure calls the conditional waiting construct is defined at a higher level in the hierarchy of clauses, then evaluation of the conditions will occur before entry to the atomic clause containing the procedure calls.

$$
\begin{aligned}
&\mathit{Action} ::= \\
&\quad \mathit{Action}\ ;\ \mathit{Action} \\
&\quad |\ \textbf{if}(\mathit{Guard})\ \textbf{then}\ \mathit{Action}\ \textbf{endif} \\
&\qquad (\textbf{elseif}(\mathit{Guard})\ \textbf{then}\ \mathit{Action}\ \textbf{endelseif}\ )^* \\
&\qquad [\textbf{else}\ \mathit{Action}\ \textbf{endelse}] \\
&\quad |\ \textbf{while}(\mathit{Guard})\ \textbf{then}\ \mathit{Action}\ \textbf{endwhile} \\
&\quad |\ v := E \\
&\quad |\ [v :=]\ m.pn(a_1, \ldots, a_k) \\
&\quad |\ v :=\ C.\textbf{create}(a_1, \ldots, a_k)
\end{aligned}
$$

The sequence operator used in *Action* is different to that of *NonAtomic* since it does not indicate a location for interleaving. Rather, the atomic sequence operator is used for updates within a transactional clause and is not visible externally.

8.2.2 The Mapping to Event-B

The definition of the mapping once again uses the Guarded Command Language of[52], as used in Chapter 3; atomic regions are bracketed thus, $\triangleleft\ \triangleright$, as before and the TNA translations are as defined in Chapter 4

The TLA mapping for Synchronized-OCB is shown in Definition 3.11, where the TLA mapping gives rise to an event, but the OCB actions require further 'treatment' when compared to Synchronized-OCB. At this point in the discussion we see the introduction of more features that differentiate Synchronized-OCB and Transactional-OCB. At the level of the *Action* clause we introduce some new concepts: Firstly we introduce direct access to attributes, which means there will be a change to the renaming function TV of section 4.1. Recall that attribute names used in the OCB actions are renamed with respect to the caller using the TV mapping function of Table 4.1. Actions now need to accommodate attributes of the form $id_1.id_2$ where id_1 is an attribute of type $MName$, representing a *SharedClass* instance known to the class, and id_2 is an attribute name belonging to that instance. The mappings defined in Table 4.1 are similar but we now accommodate a compound identifier which uses dot-notation in v. Table 8.1 supersedes the TV translation for the variable v, as shown in Table 4.1. The set of attributes vp are all of the attributes visible from the instance s. Currently all attributes have public scope by default, that is, they are navigable from any class; however we may wish to restrict this in future work by introducing a private scope, so vp will contain only the set of attributes navigable from a particular class.

$identifier\ v$	$< v,\ vp,\ s >^{TV}$	
$id_1.\cdots.id_n$	$i_n(i_{n-1}\,..\,(i_1(s)))$	where $id_1.\cdots.id_n \in vp$
	$id_1.\cdots.id_n$	where $id_1.\cdots.id_n \notin vp$
	$\forall s \cdot s \in P \wedge$ $s \in dom(id_1) \wedge$ $id_1(s) \in dom(id_2) \wedge$ $id_2(id_1(s)) \in dom(id_3)\ \wedge ..\wedge$ $id_{n-1}(id_{n-2}\ldots(id_1(s)) \in dom(id_n)$	where P is the calling process add well-definedness invariant
	$id_i[v] = id_i(s)(< v, vp, s >^{TV})$	where id_i is an array access indexed by v

TABLE 8.1: Rule TV applied to v

In OCB, when accessing an attribute, we wish to ensure that the object exists but any of the attributes of the compound identifier are potentially undefined. In the implementation if an attempt is made to access an object that does not exist, a null pointer exception would be thrown; and we can avoid this by proving, in advance, that the partial function is well-defined. To facilitate this we add a well-definedness invariant to the Event-B machine which is defined in Table 8.1. In Synchronized-OCB we simply add an event to handle the case where a call is made to a null target. We however think the proof approach used here is an improvement on the approach used in Synchronized-OCB, and could indeed be used there too. We now show an example of the mapping of a compound identifier to Event-B, where w, y and z are attributes used in the compound identifier, and vp is the set of attributes visible to the instance s.

$$\begin{aligned} &< w.y.z,\ vp,\ s >^{TV} \\ &= \\ &\quad z(y(w(s))) \end{aligned}$$

Table 8.1 also defines the mapping of an array reference. We previously discussed mapping array accesses in 4.1.1, and to recap, we defined the mapping as follows,

$$< v[i],\ vp,\ s >^{TV} = v(s)(< i,\ vp,\ s >^{TV})$$

In the application of TV to the OCB fragment the variable part and the index part of an array access is split into two parts. In the resulting Event-B fragment we have a function application v parameterised by instance s, this is then composed functionally with the translated index i. In Transactional-OCB's mapping of array accesses we split the variable and index parts in the same way, We can also see that if y was an array access $y[i]$, indexed by attribute i, and we had a compound identifier involving an array

access $w.y[i].z$, then $< w.y[i] >^{TV}$ would yield the following,

$$< w.y[i] >^{TV} = y(w(s))(< i,\ vp,\ s >^{TV})$$

and so for the complete identifier, we have,

$$< w.y[i].z >^{TV} = z(y(w(s))(< i,\ vp,\ s >^{TV}))$$

and thus we accommodate array accesses in compound identifiers.

The next feature of Transactional-OCB to consider is that of procedure calls in OCB actions. In the Synchronized-OCB definition actions were simply copied to event actions with the variables renamed. In Transactional-OCB actions we allow multiple procedure calls and create calls, these are expanded in-line where they occur in an action, and the variables are renamed. The easiest way to define the mapping is to modify the mapping of the labelled guarded action as defined in 4.4, we use P to denote the class in which the labelled clause is defined,

So the labelled guarded action is,

Definition 8.1. $< l1 :\ \triangleleft\ g \rightarrow a\ \triangleright\ ,\ l2,\ P >^{TLA}$
$\triangleq$
$l1_{\mathrm{P}} =$
ANY s
WHERE $s \in P_{\mathrm{inst}} \wedge P_{\mathrm{pc}}(s) = l1 \wedge < g,\ vp,\ s >^{TV}$
THEN $< a,\ vp,\ s,\ P,\ l1 >^{TA}\ ;\ P_{\mathrm{pc}}(s) := l2$
END

where vp is the set of variable names visible to class P.

We now have two translation functions, TA defined in Table 8.2, and $TA2$ defined in Table 8.3. TA maps a simple action to an Event-B action. TA delegates the more complex task of procedure calls and constructor calls to $TA2$. In a constructor call, for instance, we add an additional parameter to represent the new instance. We also need to add a guard to type the new variable and initialize them using a mapping of the **create procedure**. Procedure calls also result in a similar in-line expansion of the procedure action. Each variable involved in a procedure call must be renamed with respect to the appropriate object, i.e. parameters are substituted and renamed with respect to the caller. Other variables accessed in the procedure body belong to the target, and are renamed with respect to the target instance. We define TA as follows,

Definition 8.2. $TA \in Action \times \mathbb{P}(VarName) \times EventBLVar \times CName \times Label$
$\rightarrow EventBAction$

Action is the action to be mapped, $\mathbb{P}(VarName)$ is a set of variable names of the process class being mapped, $CName$ is the process class name, *label* is the next label in the sequence of labelled clauses, or the terminating label if there is none. Table 8.2 contains the mapping of simple actions using TA, they map directly to the new Event-B constructs after variable renaming,

Action	$< Action,\ vp,\ s,\ P,\ l1 >^{TA}$
$Action_1$; $Action_2$	$< Action_1,\ vp,\ s,\ P,\ l1 >^{TA}$; $< Action_2,\ vp,\ s,\ P,\ l1 >^{TA}$
if($Guard$) **then** $Action$ **endif**	**if**($< Guard,\ vp,\ s >^{TV}$) **then** $< Action,\ vp,\ s >^{TV}$ **endif**
⋮	⋮
elseif($Guard$) **then** $Action$ **endelseif**	**elseif**($< Guard,\ vp,\ s >^{TV}$) **then** $< Action,\ vp,\ s >^{TV}$ **endelseif**
⋮	⋮
else $Action$ **endelse**	**else** $< Action,\ vp,\ s >^{TV}$ **endelse**
while($Guard$) **then** $Action$ **endwhile**	**while**($< Guard,\ vp,\ s >^{TV}$) **then** $< Action,\ vp,\ s >^{TV}$ **endwhile**
$identifier := E$	$< identifier,\ vp,\ s >^{TV} :=$ $< E,\ vp,\ s >^{TV}$
$identifier :=$ Q.**create**$(a_1, \ldots, a_k)$	$< identifier :=$ Q.**create**$(a_1, \ldots, a_k),\ vq,\ vp,$ $s,\ new_q,\ P,\ l1 >^{TA2}$
$identifier :=$ M.**create**$(a_1, \ldots, a_k)$	$< identifier :=$ M.**create**$(a_1, \ldots, a_k),\ vm,\ vp,$ $s,\ new_m,\ P,\ l1 >^{TA2}$
$identifier :=$ $id.pn(a_1, \ldots, a_n)$	$< identifier :=$ $id.pn(a_1, \ldots, a_n),\ vq,\ vp,$ $s,\ j,\ P,\ l1 >^{TA2}$

where: s represents the calling instance.
P represents the class that contains the clause $l1$.
new_q represents a new process instance.
vq is the set of variable names of Q.
vp is the set of variable names of P.
new_m represents a new passiveClass instance.
vm is the set of variable names of M.
j represents the target instance.

TABLE 8.2: Rule TA

It now remains to define the mapping function for procedure and create calls, $TA2$.

The definition is similar to TA except for the addition of a parameter $EventBLvar$ representing the new instance or target, and a set of variable names $\mathbb{P}(VarNames)$ of the new instance; or target in the case of a procedure call.

Definition 8.3. $TA2 \in Action \times \mathbb{P}(VarNames) \times \mathbb{P}(VarNames) \times EventBLVar \times EventBLVar \times CName \times Label \rightarrow EventBAction$

For mappings associated with $TA2$ see Tables 8.3, 8.4, and 8.5. We separate the mapping of the procedure call and constructor constructs into separate tables since each contributes to the parameters, guard and action of the event. For each action involving a process create call we add a unique parameter new_q to the set of parameters of the event. The mapping of constructors for processes and shared classes are slightly different so we consider each in turn beginning with the process constructor, The shared class

Add	$< v := Q.\mathbf{create}(a_1, \ldots, a_k),\ vq,\ vp,\ s,\ new_q,\ P,\ l1 >^{TA2}$
Event parameter	new_q
Event Guard	$new_q \in Q_{\text{set}} \setminus Q_{\text{inst}}$
Event Action	$< a',\ vq,\ new_q >^{TV}\ ;\ Q_{\text{inst}} := Q_{\text{inst}} \cup \{new_q\}\ ;$ $v(s) := new_q\ ;\ Q_{\text{pc}}(new_q) := sLabel(na)$

where: s represents the calling instance, and new_q represents the new instance.
vq is the set of variable names of Q, and vp is the set of variable names of P.
$Q.create(f_1, \ldots, f_k) = a$
$a' = a[fn_1, \ldots, fn_k \backslash < a_1,\ vp,\ s >^{TV}, \ldots, < a_k,\ vp,\ s >^{TV}]$
na is the non-atomic clause of process Q, and $sLabel$ is defined in definition 3.6

TABLE 8.3: Rule TA2 for a Process Constructor

constructor is identical except there will be no program counter set for the new instance, and is shown in 8.4. The last action we consider is the procedure call, and is shown in 8.5

8.3 Examples Mapped to Event-B

8.3.1 The Sequential Operator within a Transactional Clause

In the following examples we make use of a shared class M with two instances *m1* and *m2*, and a *ProcessClass* P containing the labelled clauses which refer to the shared classes. We describe how various OCB features map to Event-B, and we conclude the section by drawing together the examples to show the complete OCB specification. In the first example we illustrate the ability to update multiple objects in a single clause. We swap the values of attributes $m1.x$ and $m2.x$ atomically, using direct access. We

Add	$< v := M.\mathbf{create}(a_1, \ldots, a_k),\ vm,\ vp,$ $s,\ new_m,\ P,\ l1 >^{TA2}$
Event parameter	new_m
Event Guard	$new_m \in M_{\text{set}} \setminus M_{\text{inst}}$
Event Action	$< a', vm, new_m >^{TV}\ ;\ M_{\text{inst}} := M_{\text{inst}} \cup \{new_m\}\ ;$ $v(s) := new_m$

where: s represents the calling instance, new_m represents the new instance.
vp is the set of variable names of P, vm is the set of variable names of M.
$M.create(f_1, \ldots, f_k) = a$
$a' = a[fn_1, \ldots, fn_k \backslash < a_1,\ vp,\ s >^{TV}, \ldots, < a_k,\ vp,\ s >^{TV}]$

TABLE 8.4: Rule TA2 for a Shared Class Constructor

Add	$< v := id.pn(a_1, \ldots, a_n),\ vq,\ vp,$ $s,\ j,\ P,\ l1 >^{TA2}$
Event Action	$< a',\ vj,\ j >^{TV}$
Invariant Clause	$\forall s \cdot s \in P \wedge s \in dom(P_{\text{pc}})$ $P_{\text{pc}} = l1 \Rightarrow$ $s \in dom(id)$

where: id refers to an instance, and may be a compound identifier,
s represents the calling instance, and j represents the target instance,
$id.pn(a_1, \ldots, a_k) = a$
vp is the set of variable names visible to class P.
vj is the set of variable names visible to the target instance, id,
$a' = a[fn_1, \ldots, fn_k \backslash < a_1,\ vp,\ s >^{TV}, \ldots,$
$< a_k,\ vp,\ s >^{TV}][\mathbf{return} \backslash v]$

TABLE 8.5: Rule TA2 for a Procedure Call

use an attribute *t2* of P to temporarily store the value of $m2.x$. Note that the Event-B action makes use of a sequentially composed clause, facilitated by our proposal to introduce the sequence operator into the Event-B syntax,

$$< l1 :\ \triangleleft\ t2 := m2.x\ ;\ m2.x := m1.x\ ;\ m1.x := t2\ \triangleright\ ,\ l2,\ P >^{TLA}$$

This maps to the event,

$$
\begin{array}{l}
l1_Q \triangleq \\
\textbf{ANY } s \\
\textbf{WHERE } s \in P_{\text{inst}} \ \wedge \ P_{\text{pc}}(s) = l1 \\
\textbf{THEN } t2(s) := x(m2(s)) \ ; \ x(m2(s)) := x(m1(s)) \ ; \\
\qquad\qquad x(m1(s)) := t2(s) \ ; \ P_{\text{pc}}(\text{s}) := \text{l2} \\
\textbf{END}
\end{array}
$$

Together with the invariant to ensure that the references are well defined,

$$
\begin{array}{l}
\forall s \cdot s \in P_{\text{inst}} \ \wedge \ s \in dom(P_{\text{pc}}) \ \wedge \ P_{\text{pc}}(s) = l1 \\
\Rightarrow \\
s \in dom(m2) \ \wedge \ m2(s) \in dom(x) \ \wedge \\
s \in dom(m1) \ \wedge \ m1(s) \in dom(x)
\end{array}
$$

8.3.2 Branching in a Transactional Clause

In Transactional-OCB we retain the non-atomic *if* statement of Synchronized-OCB which is defined as choice between branching clauses; this may contain further branches using additional *else* clauses. In each branch the atomic part of the clause may be followed by another non-atomic construct permitting interleaving with other processes. In Transactional-OCB we introduce a branch construct which consists of a single atomic clause; with no opportunity to interleave between the if and $else$ branches. This atomic branching clause gets mapped directly to the proposed Event-B, atomic branching clause.

In the following example we show a clause labelled *l2* which transfers an amount, *amt*, from *m1.x* to *m2.x* if $m1.x - amt \geq 0$, else makes no changes. We specify the clause using two procedure calls which gives rise to an event with the procedure bodies in-lined, formal parameters will be replaced with actual parameters and variables are renamed. The *add* procedure adds an amount a to a value x,

$$\textbf{Procedure } add(Integer\ a)\{\ x := x + a\ \}$$

The *sub* procedure subtracts an amount from x and returns the new balance,

$$\textbf{Procedure } sub(Integer\ a)\{\ x := x - a\ ;\ \textbf{return} := x\ \}\ :\ Integer$$

In the current example the return value of *sub* can be ignored since there is no assignment involved, but we will use this in a later example. The following procedure *getx* has no side-effects, so we are able to use it in an expression and simply substitute in-line, and

rename using TV, the RHS of the return statement.

$$\textbf{Procedure } getx()\{\ \textbf{return} := x\ \}\ :\ Integer$$

The OCB specification to make an atomic transfer of an amount amt follows,

$$< l2 :\ \triangleleft\ \textbf{if}(m1.getx() - amt \geq 0)\ \textbf{then}$$
$$m1.sub(amt)\ ;\ m2.add(amt)\ \textbf{endif}\ \triangleright\ , l3,\ P >^{TLA}$$

This maps to the following,

$$\begin{array}{l}
\textbf{ANY } s \\
\textbf{WHERE } s \in P_{\text{inst}}\ \wedge\ P_{\text{pc}}(s) = l2 \\
\textbf{THEN if}(x(m1(s)) - amt(s) \geq 0)\ \textbf{then} \\
\qquad\qquad x(m1(s)) := x(m1(s)) - amt(s)\ ; \\
\qquad\qquad x(m2(s)) := x(m2(s)) + amt(s)\ \textbf{endif}\ ; \\
\qquad\quad P_{\text{pc}} := l3 \\
\textbf{END}
\end{array}$$

and we add the following invariant to ensure that the references are well defined,

$$\begin{array}{l}
\forall s \cdot s \in P_{\text{inst}}\ \wedge\ s \in dom(P_{\text{pc}})\ \wedge\ P_{\text{pc}}(s) = l2 \\
\Rightarrow \\
s \in dom(m2)\ \wedge\ m2(s) \in dom(x)\ \wedge \\
s \in dom(m1)\ \wedge\ m1(s) \in dom(x)\ \wedge \\
s \in dom(amt)
\end{array}$$

The branching clause will give rise to a proof obligation, based on that of the classical-B [5] branching construct. We ensure that a post condition P is established after the branch; so for a branch **if** g **then** a_1 **else** a_2 **endif**; we show that the following holds,

$$(g \wedge [a_1]P) \vee (\neg g \wedge [a_2]P)$$

8.3.3 Looping in a Transactional Clause

The following loop accesses variables using a procedure call to set the value of an attribute x. We decrement x of *m1*, and increment x of *m2*, i times, using a procedure *setx* (defined as follows) to set the value of x,

$$\textbf{Procedure } setx(Integer\ v)\{\ x := v\ \}$$

We assign the value of $m1$'s x attribute to $t1$ after decrementing it, and then we assign the value of $m2$'s x attribute to $t2$ after incrementing it. These are passed as parameters to the *setx* procedure calls,

$$< l3 : \ \triangleleft \ \textbf{while}(i > 3) \ \textbf{do}$$
$$t1 := m1.getx() - 1 \ ; \ t2 := m2.getx() + 1 \ ;$$
$$m1.setx(t1) \ ; \ m2.setx(t2) \ ; \ i := i - 1 \ \textbf{endwhile} \ \triangleright \ , \ l4, \ P >^{TLA}$$

We show the mapping to Event-B now where $l4$ is supplied as the next label parameter,

$$\textbf{ANY} \ s$$
$$\textbf{WHERE} \ s \in P_{\text{inst}} \ \wedge \ P_{\text{pc}}(s) = l3$$
$$\textbf{THEN while}(i(s) > 3) \ \textbf{do}$$
$$t1(s) := x(m1(s)) - 1 \ ; \ t2(s) := x(m2(s)) + 1 \ ;$$
$$x(m1(s)) := t1(s) \ ; \ x(m2(s)) := t2(s) \ ;$$
$$i(s) := i(s) - 1 \ \textbf{endwhile} \ ;$$
$$P_{\text{pc}} := l4$$
$$\textbf{END}$$

and we add the following invariant to ensure well-definedness of attributes,

$$\forall s \cdot s \in P_{\text{inst}} \ \wedge \ s \in dom(P_{\text{pc}}) \ \wedge \ P_{\text{pc}}(s) = l3$$
$$\Rightarrow$$
$$s \in dom(t1) \ \wedge \ s \in dom(t2) \ \wedge$$
$$s \in dom(m1) \ \wedge \ m1(s) \in dom(x)$$
$$s \in dom(m2) \ \wedge \ m2(s) \in dom(x) \ \wedge$$
$$s \in dom(i)$$

Treatment of proof obligations for loops in OCB is based on that of the classical-B looping construct. It may be necessary to prove loop that a loop terminates - that is, it does not continue looping forever. To do this we add a natural number loop variant and show that the loop always decrements the variant. We add a loop variant to a clause in the following manner,

$$\textbf{while} \ g \ \textbf{do} \ a$$
$$\textbf{variant} \ V \ \textbf{endwhile}$$

A proof obligation is then generated which must be discharged to show that the variant decreases during each iteration. So for any variant V, and its updated value V', we must show that $V' < V$. Now, the variant V is a natural number, and it must be the

case that the loop eventually terminates since a natural number cannot be decremented forever. In addition it is also necessary to show that, after each loop iteration, that the loop invariant I holds, which is shown by $I \wedge g \Rightarrow [a]I$. We therefore show that if the loop invariant I and the loop condition g holds initially then the action a should re-establish the invariant. Finally, if the loop condition no longer holds, that is $\neg g$, then some postcondition P is shown to hold. The complete proof obligation follows, where x is the list of state variables that can be changed in the loop body.

$$
\begin{aligned}
& I \wedge \\
& \forall x \cdot (I \wedge g \Rightarrow [a]I) \wedge \\
& \forall x \cdot (I \Rightarrow V \in \mathbb{N}) \wedge \\
& \forall x \cdot (I \wedge g \wedge V = V' \Rightarrow [a](V' < V)) \wedge \\
& \forall x \cdot (I \wedge \neg g \Rightarrow P)
\end{aligned}
$$

8.3.4 Procedure Bodies

We extend the use of the new constructs to procedures. A procedure body can make use of the new sequential, branching and looping constructs. We present an example that uses a while loop in the procedure body, and show its mapping to Event-B. The procedure call *iter* is defined in a class M which contains a loop,

$$\textbf{Procedure } iter()\{ \textbf{ while}(c > 0) \textbf{ do } x := x + 1 \ ; \ c := c - 1 \textbf{ endwhile } \}$$

In the following labelled clause $m1$, an attribute of type M, is used to call M's *iter* procedure,

$$< l4 : \ \triangleleft \ m1.iter() \ \triangleright \ , \ terminated, \ P >^{TLA}$$

This maps to,

$$
\begin{aligned}
& \textbf{ANY } s \\
& \textbf{WHERE } s \in P_{\text{inst}} \wedge P_{\text{pc}}(s) = l4 \\
& \textbf{THEN while}(c(m1(s)) > 0) \textbf{ do} \\
& \qquad\qquad x(m1(s)) := x(m1(s)) + 1 \ ; \\
& \qquad\qquad c(m1(s)) := c(m1(s)) - 1 \textbf{ endwhile } ; \\
& \qquad P_{\text{pc}}(\text{s}) := terminated \\
& \textbf{END}
\end{aligned}
$$

and we add a well-definedness invariant,

$$\forall s \cdot s \in P_{\text{inst}} \ \wedge \ s \in dom(P_{\text{pc}}) \ \wedge \ P_{\text{pc}}(s) = l4$$
$$\Rightarrow$$
$$s \in dom(m1) \ \wedge \ m1(s) \in dom(x) \ \wedge \ m1(s) \in dom(c)$$

We show the complete specification in Figure 8.2, notice the use of brackets $< \ldots >$ to enclose atomic clauses in the textual specification, this corresponds to out use of $\triangleleft \ldots \triangleright$ in the rule definitions.

```
ProcessClass P{
 M m1, M m2, Integer i, Integer t1, Integer t2, Integer amt
 Procedure create(M ma, M mb, int amount){
   m1:=ma; m2:=mb; i:=6; t1:=0; t2:=0; amt:=amount}
 Operation run(){
  l1: <t2 := m2.x ; m2.x := m1.x ; m1.x := t2>  ;
  l2: <if(m1.getx()-amt >= 0) then m1.sub(amt) ; m2.add(amt) endif> ;
  l3: <while(i>3) do t1:=m1.getx()-1 ; t2:=m2.getx()+1 ;
       m1.setx(t1) ; m2.setx(t2) ; i := i-1 endwhile> ;
  l4: <m1.iter(i)>
 }
}

SharedClass M{
 Integer x, Integer c
 Procedure create(){x := 0 ; c := 3}
 Procedure getx(){return := x}:Integer
 Procedure setx(Integer v){x := v}
 Procedure add(Integer a){x := x+a}
 Procedure sub(Integer a){x := x-a;return := x}:Integer
 Procedure iter(){ while(c>0) do x:=x+1 ; c:=c-1 endwhile}
}
```

FIGURE 8.2: A Transactional-OCB Specification

8.4 Mapping to Java

In OCB specifications, and the corresponding Event-B actions, we use sequential composition. Therefore the mapping to Java is straightforward, and most OCB actions can be mapped directly to Java statements. Both the OCB atomic and non-atomic semi-colon operators map directly to the Java semi-colon delimiter; the complexity lies in making the action transactional. That is, we require a method of isolating the actions of one process from the actions of other processes in order to prevent interference. If a

transaction cannot progress (perhaps a lock cannot be acquired) then no changes should be visible to other processes - no partial updates must be seen by other processes.

We previously mentioned the STM approach to implementing transactions, and we may investigate STM as an alternative to a lock-based scheme in the future. We give an overview of an STM implementation in Section 8.4.4, but continue our discourse using our own lock-based implementation using some of Java 1.5's new concurrency features.

8.4.1 Locking

Before elaborating on the mapping to Java we need to discuss locking of objects in some detail. We describe a locking scheme so that the objects accessed in an OCB clause will be locked in the corresponding Java implementation to allow mutually exclusive updates. We also have a similar conditional locking feature to Synchronized-OCB, but a *when* clause guards a whole labelled transactional clause. The conditional critical region may therefore contain a sequence of assignments and procedure calls which are executed atomically, i.e. the contents of a labelled transactional clause. This is in contrast to the conditional critical region of Synchronized-OCB, which is specified wholly within the procedure body of an atomic procedure call. In an OCB specification there is no explicit notation added to wake blocked processes, rather the translator adds the appropriate calls to wake waiting processes. In Java 1.5 threads are caused to wait by calling a *Condition* object's *await* method. They are woken by calling a *condition* object's *signalAll* or *signal* method. To wake threads we insert a *signalAll* call near to the end (prior to lock release) of methods which make updates to shared class variables. The blocking and signalling of threads is hidden from the developer in order to simplify reasoning about the behaviour of the system. We now summarize the steps for ensuring mutually exclusive access to shared objects accessed in labelled atomic clauses,

- A process locks each shared object accessed in the clause.
- If the clause is guarded - check the entry conditions
  - If an entry condition is not satisfied for some shared class, then block waiting for its data to be updated after releasing appropriate locks ; return to the first step upon waking.
  - Continue to critical region if entry conditions are satisfied.
- Perform updates in the critical region.
- Release locks

Each transactional clause can refer to one or more instances, these instances are identified by attributes, declared in the class in which the clause is used. To prevent interference

by other processes a process must own the locks of the instances it accesses before reading or updating variables. We use a non-blocking approach to lock acquisition where each labelled clause is associated with a number of locks. Since shared classes may contain other shared classes (a parent-child relationship) it is not possible to know, before obtaining a parent class' lock, which instance (the child) a particular attribute refers to. This is because if a process does not hold a parent's lock, some other process is free to change the object that the attribute refers to. It is for this reason that it is necessary to obtain each parent's lock before ascertaining which instance its attributes (children) refer to; and therefore which instances needs locking. In the event of failure to acquire a lock, previously acquired locks are released and the thread remains active. The locking activity takes place within a loop that only terminates when all locks have been acquired. Releasing previously acquired locks, in the event of a failure to acquire a lock, alleviates the problem of deadlock due to resource retention.

We demonstrate the locking policy using an example. Given an OCB clause containing access to an attribute $a.b$ where b is an instance of some class, we say that b is nested in a; and later we refer to the top-level with a as level 1, and the second level with b as level 2, and so on. When we want to acquire the top-level lock a we already know which instance a refers to - the process class is not shared so it cannot be changed by any other process - so we try to obtain the lock. If successful we then attempt to obtain the lock of the next level, b. Now since a is a shared object b can be changed by another process; but the calling process owns a's lock, so no other process will be able to access b and interfere with it. Lock acquisition proceeds in this way, traversing the lock levels, until all locks are successfully acquired. It may be the case that instead of using direct access a procedure call may be used to access b, such as $a.getB()$ instead of $a.b$. In such circumstances, as long as the call has no side-effects (to be checked statically), we proceed in the same way as with direct access; until we encounter a procedure call. We would then invoke the getter method to obtain the object, and continue as before by attempting to obtain that object's lock.

A *ProcessClass* maps to a Java class implementing the *java.lang.Runnable* interface. The *run* method is populated by method calls derived from the OCB labelled transactional clauses of the OCB *run* operation. Each of the *run* operation's clauses is linked to a method call, and a corresponding method declaration implementing the required behaviour.

```
public void run() {
// Sequence of atomic clause calls
  l1();
  l2();...
}
```

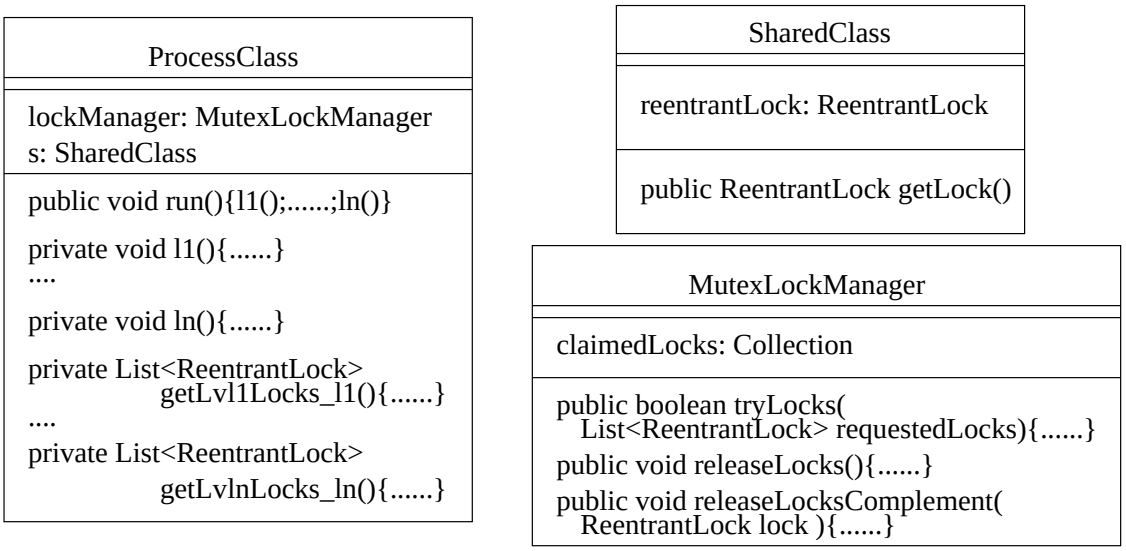


FIGURE 8.3: The LockManagement Structure

```
 // Implementations of atomic clauses
 public void l1() { ... }
 public void l2() { ... }...
}
```

To implement the mutual exclusion policy each *SharedClass* instance has a lock provided by *java.util.concurrent.locks.ReentrantLock*. We use its *trylock* and *unlock* methods to lock and unlock the object. Each labelled transactional clause is related to one or more helper methods that returns a collection of locks. These methods acquire the locks of a given nesting level for each clause, so $getLvl1Locks\_l1$ is a getter method for the level 1 locks of the clause labelled $l1$. All of a clause's required locks, at each level, must have been claimed before entry to the critical region is possible. We show a diagram of the classes involved in Figure 8.3. The collections returned by the helper methods are used by a lock manager (an instance of *MutexLockManager*) to co-ordinate lock acquisition and release. There is one lock manager instance per labelled transactional clause. The *MutexLockManager* class has a method *tryLocks* to obtain locks, this calls the *ReentrantLock*'s *tryLock* method for each lock in the collection; and releases the locks contained in the *claimedLocks* collection in the event of failure to acquire any lock. The *MutexLockManager* class also has a *releaseLocks* method to release all of the locks in the *claimedLocks* collection. The $MutexLockManager$ class is shown in Figure 8.4. Successfully claimed locks are stored in a collection, *claimedLocks*. We now extend the OCB model of Figure 8.2, to demonstrate our approach to locking objects referred to by a compound identifier. To the existing model we add the following shared class with an integer attribute w,

$$\textbf{public class } N\{\textbf{public int } w = 0;\}$$

```
public class MutexLockManager {
 private List<ReentrantLock> claimedLocks =
       new ArrayList<ReentrantLock>();

 public boolean tryLocks(List<ReentrantLock> requestedLocks){
  for(int i = 0; i<requestedLocks.size();i++){
   ReentrantLock lock = requestedLocks.get(i);
   if(lock.isHeldByCurrentThread()) claimedLocks.add(lock);
   else{
    if(lock.tryLock())claimedLocks.add(lock);
    else{
     if(claimedLocks.size()>0) releaseLocks();
     return false;
    }
   }
  }
  return true;
 }

 public void releaseLocks(){
  while(claimedLocks.size()>0){
   ReentrantLock lock = claimedLocks.get(0);
   lock.unlock();
   claimedLocks.remove(lock);
  }
 }
...
```

FIGURE 8.4: Part of the MutexLockManager Class

We add a clause $l5$ to the process class *run* method, see Figure 8.5. The clause simply increments the attribute w of shared class n, but has to first acquire the lock of $m1$ and then n.

$$l5 : m1.n.w := m1.n.w + 1$$

In any implementation we must first acquire the top-level lock, in this case that of $m1$ - and then acquire n's lock. To do this we use helper the methods, $getLevel1Locks\_l5$ and $getLevel2Locks\_l5$ (shown in Figure 8.6) to return required locks for each level, and pass the resulting collection to the *tryLocks* method. The helper methods are created as part of the translation process.

We now explore the use of procedure calls to specify the equivalent clause. In the following we can envisage getter procedures to obtain the values of n and w used in the following way,

$$l5 : m1.getN().getW() := m1.getN().getW() + 1$$

Here, $m1.getN()$ obtains the object associated with attribute n belonging to $m1$; in turn the $getW$ procedure obtains the value of attribute w associated with the object

```
ProcessClass P{
 M m1, M m2, Integer i, Integer t1, Integer t2, Integer amt
 Procedure create(M ma, M mb, int amount){
    m1:=ma; m2:=mb; i:=6; t1:=0; t2:=0; amt:=amount}
 Operation run(){
 ...
  l5: <m1.n.w := m1.n.w + 1>
 }
}

// This class is nested in M
public class N{
 public int w = 0 ;
}

// This class contains class N
SharedClass M{
 Integer x, Integer c, N n
 Procedure create(N ne){x := 0 ; c := 3; n:= ne}
 ...
}
```

FIGURE 8.5: A Transactional-OCB Specification with a Compound Identifier

referenced by n. In this situation we must ensure that the getter procedures are side-effect free. So the getter procedures for n and w would simply access an attribute and return its value. However, if side-effects were allowed then the lock manager would have to perform the processing prior to obtaining the lock (at the lock acquisition stage) and therefore it may cause updates to state before entry into the critical region which is not allowed. One could also envisage the situation where updates, caused by a call to a procedure causing side-effects, would have to be rolled back if a subsequent lock is not available. Rollback behaviour in these circumstances is beyond the scope of this thesis. The mapping of the OCB procedure calls to Java code is a straightforward copy with the *becomesequal* operator substituted for assignment.

In order to implement the locking strategy we place *tryLocks* and *releaseLocks* calls in appropriate positions in the resulting code. An example of this can be seen in our extension of process P, in Figure 8.6, where we acquire the locks of nested objects related to the clause labelled $l5$.

8.4.2 Translating Transactional Clauses to Java

We can say that much of the translation from an OCB specification to Java proceeds as for the Synchronized-OCB translation; until we consider the translation of non-atomic constructs and labelled atomic clauses. The translation is similar to the translation

```
public class P implements Runnable{
 protected MutexLockManager lockManager= new MutexLockManager();
 // variables belonging to the process
 private M1 m1; ...
 ⋮
 private List<ReentrantLock> getLevel1Locks_l5(){
  List<ReentrantLock> level1Locks_l5= new ArrayList<ReentrantLock>();
  level1Locks_l5.add(m1.getLock());
  return level1Locks_l5;
 }

 private List<ReentrantLock> getLevel2Locks_l5(){
  List<ReentrantLock> level2Locks_l5= new ArrayList<ReentrantLock>();
  level2Locks_l5.add(m1.n.getLock());
  return level2Locks_l5;
 }

 // the process calls the method related to the labelled transaction
 public void run(){ ... l5(); }
 // compound id example, m1.n.w := m1.n.w + 1
 private void l5(){
  // acquire locks
  boolean hasLvl1Locks = false;
  boolean hasLvl2Locks = false;
  // any failure will reset level 1 locks
  // no failure allows progress
  while (!hasLvl1Locks){
   // get level1 locks
   hasLvl1Locks = lockManager.tryLocks(getLevel1Locks_l5());
   // if successful get level 2 locks
   if (hasLvl1Locks){
    while (!hasLvl2Locks){
     hasLvl2Locks = lockManager.tryLocks(getLevel2Locks_l5());
    }
    // if level2 (or deeper) cannot be obtained, any claimed locks
    // will have been released, so reset the level1 lock flag too
    if (!hasLvl2Locks) hasLvl1Locks = false;
   }
  }
  //ENTER CCR
  m1.n.w = m1.n.w + 1;
  //EXIT CCR
  lockManager.releaseLocks();
  hasLvl1Locks=false;
  hasLvl2Locks=false;
 }
}
```

FIGURE 8.6: The Locking of Nested Shared Classes

defined in Table 4.4, except now a non-atomic construct may give rise to one or more method declarations, and statements that invoke those methods placed in appropriate locations in the code. The labelled transactional clauses are translated to Java by applying the translation function $ltDef$ of Figure 8.7. The modified translation $naDef$ is shown in Table 8.6, and for now we ignore conditional waiting. We will try to simplify the explanation of the translation as much as possible. In order to simplify the description of the transactional constructs we provide a pseudo methods, $\ldots acquire$, $\ldots release$ to describe the location of locking acquisition and release code.

The first step in translation of a non-atomic construct to Java code is to apply the translation function $naDef$, with the non-atomic passed as a clause parameter. This gives rise to, for each labelled clause in the non-atomic construct, one or more Java method declarations in the class body, and one or more method calls invoking these methods, to perform the behaviour specified in the labelled clause. The looping construct, for instance, gives rise to an atomic method, implementing the condition check and associated action - followed by other atomic methods that implement the any other atomic clauses on the 'true' branch. If the condition is false the loop quits without performing updates. The branching construct atomically evaluates the condition and performs any updates for the appropriate branch; a variable, local to the specifying class, records which branch was executed and later uses this to invoke the appropriate branch when executing the non-atomic part of the construct.

Now we discuss the translation of labelled transactional clauses l_i, which are composed using the non-atomic construct of na. The translation to Java gives rise to two statements; a method call $l_i()$, and a method declaration. The method declaration arises from the application of the translation function $ltDef$ to the labelled transactional clause. During the translation $ltDef$ adds method calls to the method body, to obtain and release the locks. The added method calls are to the $tryLocks$ and $releaseLocks$ methods described earlier in Section 8.4.1. The translation of Transactional-OCB atomic actions is shown in Definition 8.7. where a is an *Action*.

$$
\begin{aligned}
&< l1 : a >^{ltDef} \\
&= \\
&\textbf{public void } l1(\{ \\
&\quad \ldots acquire\ ; \\
&\quad < a >^{aDef} ; /{*}Critical\ Region{*}/ \\
&\quad \ldots release\ ; \\
&\}
\end{aligned}
$$

FIGURE 8.7: Rule $ltDef$ for Labelled Atomic Clauses

The atomic actions map quite easily to Java and use the definition of $cDef$ of 4.7,

na	call	$< na >^{naDef}$
na_1 ; na_2		$< na_1 >^{naDef}$; $< na_2 >^{naDef}$
l_i : **if**(c_1) **then** a **andthen** na_1 **endif** [**elseif**(c_i) **then** a **andthen** na_i **endelseif**] [**else** a **andthen** na_n **endelse**]	$l_i()$ l_i_aux(b)	$l_i()\{$ *...acquire* **if**($< c_1 >^{cDef}$)$\{ < a >^{aDef}$; $b = 1;$ $\}$ [**else if**($< c_i >^{cDef}$)$\{ < a >^{aDef}$; $b = i;\}$] [**else**$\{< a >^{aDef}$; $b = n;$ $\}$] *...release* $\}$ l_i_aux(b)$\{$ **switch**(b)$\{$ **case** 1: $< na_1 >^{naDef}$; break; [**case** i: $< na_i >^{naDef}$; break;] [**case** n: $< na_n >^{naDef}$; break;] $\}\}$
l_i : **while**(c) **then** a **andthen** na **endwhile**	l_i_while(l_i_s()) l_i_f();	l_i_s()$\{$ *...acquire* **if**($< c >^{cDef}$)$\{$ $< a >^{aDef}$; *...release;* **return true**;$\}$ **else**$\{$ *...release;* **return false**;$\}$ $\}$ l_i_f()$\{$ $< na >^{naDef}$ $\}$
a		$< a >^{aDef}$;
l_i : a	$l_i()$	$< l_i : a >^{ltDef}$

TABLE 8.6: Rule naDef for Transactional Clauses

8.4.3 Conditional Waiting

It may be the case that we require some condition to be satisfied before processing of a transaction continues. It is therefore appropriate to check the condition, immediately after the locks have been acquired, and before entry into the critical region. Assume that we have specified two buffers belonging to channel c which are accessed using dot notation as follows, $c.buff1$ and $c.buff2$, and that the size is held in an attribute *size* of the buffer; we may wish to ensure that both buffers are empty before executing action

a	$< a >^{aDef}$
$a\ ;\ a$	$< a >^{aDef}\ ;\ < a >^{aDef}$
if(c) **then** a **endif** [**elseif**(c) **then** a **endelseif**] [**else** a **endelse**]	**if**$(< c >^{cDef})\{< a >^{aDef};\}$ [**else**{**if**$(< c >^{cDef})\{< a >^{aDef};\}\}$] [**else**$\{< a >^{aDef};\}$]
while(c) **then** a **endwhile**	**while**$(< c >^{cDef})\{< a >^{aDef}\}$
$identifier := E$	$identifier = E;$
$identifier.pn(a_1, \ldots, a_k)$	$identifier.pn(a_1, \ldots, a_k)$
$identifier := identifier.pn(a_1, \ldots, a_k)$	$identifier = identifier.pn(a_1, \ldots, a_k)$
$identifier := Q.\mathbf{create}(a_1, \ldots, a_k)$	$identifier =$ **new** $Q(a_1, \ldots, a_k);$ **new** Thread($identifier$).start()
$identifier := M.\mathbf{create}(a_1, \ldots, a_k)$	$identifier =$ **new** $M(a_1, \ldots, a_k);$

TABLE 8.7: Rule aDef for Atomic Actions

a, or blocking otherwise, by writing a conditional statement as follows,

$$l1 : \triangleleft\ \mathbf{when}(\ c.buff1.size = 0\ \wedge\ c.buff2.size = 0\ )\{\ a\ \}\ \triangleright$$

This can be represented as a guarded action of the type $l1 :\ \triangleleft\ g \rightarrow a\ \triangleright$, so for the above example the guarded action is,

$$l1 :\ \triangleleft\ (\ c.buff1.size = 0\ \wedge\ c.buff2.size = 0\ ) \rightarrow a\ \triangleright$$

Any labelled clause may contain a *when* clause, with a condition that guards entry to a critical region. The condition consists of a number of guards, but we stipulate that each guard is must be associated with only one shared class. In this way we are able to associate a failed condition with a particular shared class and block the thread until that particular shared class is updated. We discuss this issue in more detail later in the section, but for now we show the updated $naDef$ translation function, where we add guarded actions to the allowable na clauses of Table 8.8, We update the

na	$< na >^{naDef}$
$l1 :\ \triangleleft\ g \rightarrow a\ \triangleright$	l1() $< l1 :\ \triangleleft\ g \rightarrow a\ \triangleright\ >^{ltDef}$

TABLE 8.8: Rule naDef for a Conditional Transactional Clause

$ltDef$ translation function to accommodate the guarded clause, we use a pseudo-method

```
public void releaseLocksComplement(ReentrantLock lock){
   claimedLocks.remove(lock);
   while(claimedLocks.size()>0){
     ReentrantLock l = claimedLocks.get(0);
     l.unlock();
     claimedLocks.remove(l);
   }
}
```

FIGURE 8.8: The MutexLockManager *releaseLocksComplement* Method

acquireConditionally for simplicity, to show placement of the conditional evaluation code, and explain in more detail in the text that follows. The translation $ltDef$ of a guarded action follows,

$$
\begin{aligned}
&< l1 : g \rightarrow a >^{ltDef} \\
&= \\
&\textbf{public void } l1(\{ \\
&\quad \ldots acquireConditionally\ ; \\
&\quad < a >^{aDef} ; /{*}Critical\ Region{*}/ \\
&\quad \ldots release\ ; \\
&\}
\end{aligned}
$$

where a is an *Action*. To implement this conditional waiting approach we make use of the *java.util.concurrent.locks.Condition* API; each shared class in a guarded clause's condition has a *Condition* object associated with it. We use the *await* and *signalAll* methods associated with the condition object to block and wake threads respectively. We provide an *X_GuardManager* class for each labelled clause with label X that has a guard associated with it; the guard manager contains methods which evaluate the guards, and in the case of a false guard releases locks and blocks the calling thread. A lockManager may have acquired a number of locks, and if a condition is not satisfied all the locks should be released, that is except for the one associated with the condition we wish to wait for (we wait for the shared class associated with the failed entry condition, and the *await* method contains an implicit lock release). To unlock the locks (with the single exception noted) and remove all locks from the list of claimed locks we use the *releaseLocksComplement* method. The method *releaseLocksComplement* is shown in Figure 8.8, where we pass the lock that we do not want to unlock as a parameter, and remove this from the list of claimed locks. We then unlock and remove the remainder of the locks from the list of claimed locks. Any shared class with a condition object needs to invoke the condition object's *signalAll* method after making state updates. This will wake any threads waiting on the shared class's condition object, allowing threads to attempt to acquire the lock and re-check the entry condition. Assignment to an

attribute of a calling process is possible for procedures returning a value; this aspect is not affected by the new style conditional waiting construct.

We now return to the subject of the guards in the conditional waiting clause. We specify a Transactional-OCB clause using conditional waiting as follows, $l1 : \ \triangleleft$ **when**$(g)\{a\} \triangleright$. Here g is a conjunction of guards defined as, $g = c_1 \wedge \ldots \wedge c_k$. In the specification if g is true then a occurs, or blocks otherwise. In the implementation a method is used to evaluate each guard, c_i, in turn - and the method returns true if all the guards are true, or returns false otherwise. If one of the guards is not satisfied then the condition's *await* method is called, after calling *releaseLocksComplement*.

One restriction, which we discuss but do not attempt to resolve at this time, is where a condition refers to more than one shared class and the guard fails - which shared class should we wait for? In a condition such as $a.x < b.x$ two shared classes a and b are involved, each with an attribute x. Changes to either shared class by some other process could make the condition true, so we really need to wait for changes to either shared class. Implementing this behaviour will require additional coding effort; and we save this for future work. In the mean time we simply restrict association of each guard with a single shared class. One work-around for the moment is to perform a static check, and issue a warning when two shared classes are referred to in a conjunct; the implementation will however block on the first shared class encountered in the guard.

The guarded clause, defined above, maps to a method *l1Guard* in the *L1GuardManager* class, shown in Figure 8.9. The shared classes $m_1 \ldots m_n$ referred to in the guards are passed as parameters, this shared class instance is the one which will send the signal to wake the process, and the caller of $l1Guard$ determines which shared class to wait for.

If all guards are satisfied the calling process can proceed to the Conditional Critical Region (CCR). Figure 8.10 shows the method $l1Guard$ guarding the CCR. The CCR will not be entered until the guard returns true which enables termination of the containing loop. The *getlock* method returns the *ReentrantLock* object associated with a shared class, and *getCondition* returns its Condition object. The definition of rule $aDef$ for Transactional-OCB actions is shown in Table 8.7.

8.4.4 An Alternative: Locking Based on STMs

An alternative approach to implementing our Transactional-OCB constructs could be based on the STM implementation described in [74] which uses Java SDK 1.2. technology. The advantage of STM over a more traditional locking strategy is that the approach allows more concurrency for the non-conflicting operations, and there is a simplified approach to specifying conditional waiting. The STM approach requires a number of memory consistency rules be adhered to ensure that the integrity of the transactions

```
public boolean l1Guard(MutexLockManager l1_lockManager, M m_1, ..., M m_k){
 if(!g_1){
   l1_lockManager.releaseLocksComplement(m_1.getLock());
   try {m_1.getCondition().await();}
   catch(InterruptedException e){...}
   return false;
 }else...
 else if(!g_k){
   l1_lockManager.releaseLocksComplement(m_k.getLock());
   try{m_k.getCondition().await();}
   catch(InterruptedException e){...}
   return false;
 }else return true;}
```

where m_i is the shared class associated with guard g_i, for each $i \in 1 \ldots k$

FIGURE 8.9: Implementation of Conditional Wait in the GuardManager Class

```
Manager gm = new L1GuardManager();
...
private void l1(){
 boolean success = false;
 while(!success){
   ...acquire ;
   success = gm.l1Guard(l1_lockManager, m_1, ..., m_n);
 }
 //enter CCR
  < a >^aDef;
 //exit CCR
  ...release ;
}
```

FIGURE 8.10: Guarding the CCR with a GuardManager Instance

within the system are not violated. For instance, native code cannot be used in STM transactions apart from some very limited cases, such as cloning objects for local use.

In the STM approach the specification of an atomic region with conditional waiting is as follows,

```
atomic(condition){
   // enter CCR
   statements;
   // exit CCR
}
```

If conditional waiting is not required then the conditional part is just omitted. The atomic region is bounded by braces and *condition* is evaluated before entry into to the critical region; if false the process blocks by putting the transaction into a sleeping state.

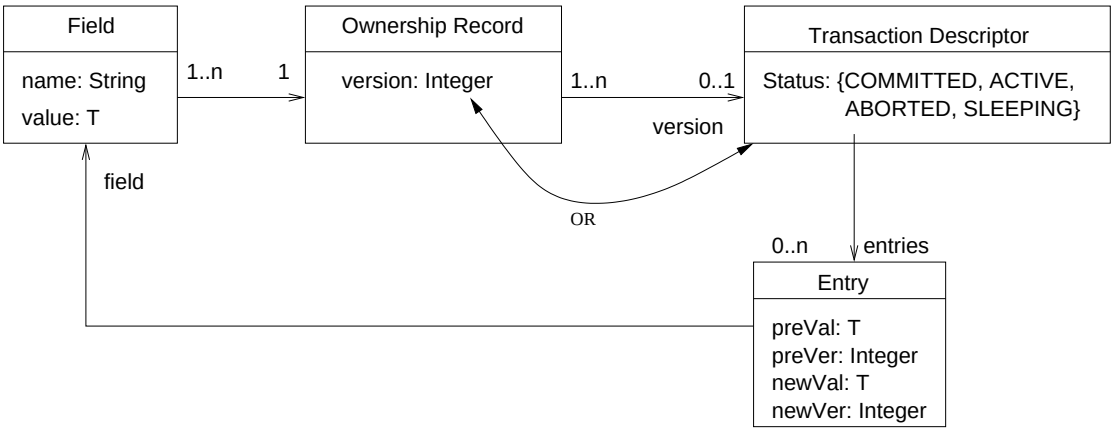


FIGURE 8.11: Class Diagram of an STM Implementation

The various transaction states can be seen in the Transaction Descriptor's *status* field in Figure 8.11. When a transaction is sleeping it is waiting to be woken when some other transaction may have updated state that the transaction refers to,that is when the ownership record associated with that state has been updated.

This STM implementation uses ownership records and transaction descriptors. The transaction descriptors keep track of the old and new values of fields accessed by a transaction, together with a versioning scheme which keeps track of the changes to individual fields. The ownership records (orecs) keep track of which variables are accessed by which transaction descriptors. There are a number of STM API methods which are used to implement the approach described in Table 8.9, which correspond to usual operations involving transactions. We now give more details of the descriptors, orecs and their

Method	Description
STMStart()	Allocates a Descriptor, Status = ACTIVE
STMAbort()	Status = ABORTED
STMCommit()	Acquire orecs, Status=COMMITTED
	update field values, release orecs
STMValidate()	Version checking of fields
STMWait()	Conditional wait, Status = SLEEPING
STMRead()	Read from existing descriptor if it exists
	else determine logical state
STMWrite()	Write new descriptor entry

TABLE 8.9: STM API Methods

relationship with fields. Figure 8.11 describes the relationships in an implementation. The fields that the transaction can refer to have a name, a value of its declared type T, and an ownership record. The ownership record stores either a version number (if no transaction has acquired the field's *orec* since it was last committed) or a transaction descriptor that has been acquired to commit changes to the field. When a transaction reads or writes to a field it will create an entry in the transaction descriptor that stores the initial value and version number in its $preVal$ and $preVer$ fields, and stores the

updated value and version number in the $newVal$ and $newVer$ fields. The new version number is obtained from an existing entry in the same transaction descriptor if one exists, or from the *orec* itself otherwise. The initial values of a field are described as its *logicalstate*, and this is derived from the information held by the system including; which field is being accessed, its associated orec, the status of any transaction descriptors that have acquired the *orec* of the field, and the various descriptor entries of those transaction descriptor. A decision procedure is applied to determine the logical state, which depends on whether the field has been accessed by another transaction since it was last committed; whether the current transaction has previously accessed the field; and also whether some other transaction involving the fields has been committed, but the field has not yet been updated in memory.

The key to the atomic update of this STM implementation resides in the status of the transaction descriptors that have accessed a particular field's orec. If a transaction descriptor changes its status to commit, it is considered to have completed the transaction even though the changes may not have been applied to the field value. This means that any other transaction, trying to commit with a field contained in an already committed transaction, must fail. The unsuccessful transaction commit will fail when it checks to see if it has the latest version of field value and finds that some other transaction has made a more recent update - that is, it has a higher version number. The check is implemented in the following *acquire* method, called by the $STMCommit$ operation. It which accepts a *descriptor* parameter, and an entry, i. During an attempted commit phase each entry i in the list of entries is checked using *acquire*. Returning $TRUE$ indicates that the *orec* for a particular field was acquired, $FALSE$ indicates failure to acquire the *orec* since the version has changed, and $BUSY$ indicates that some other transaction is already active with the orec. The following operation refers to attributes shown in Figure 8.11,

```
acquire(Descriptor descriptor, int i) {
 Entry e = descriptor.entries[i];
 Object seen ; // Holds an Integer or Transaction Descriptor
 seen = CAS(e.field.orec.version, e.preVer, descriptor);
 // Existing descriptor held or successful installed
 if(seen == e.preVer || seen == descriptor) return TRUE;
 // orec holds some other version - changed by another transcation
 else if(seen instanceof Integer)) return FALSE;
 // seen must be a transaction descriptor - so its busy
 else return BUSY;
}
```

The code of the preceding fragment attempts to install the descriptor d in the ownership record associated with the transaction entry. It makes use of the $getOrec(e.field)$

method which obtains the *orec* associated with the field in the entry e. This is then used in a CAS method (an atomic compare and swap method used to update a field atomically). The CAS method has three input parameters $currentVal$, $expectedVal$, and $newVal$; and returns $currentVal$. The method compares a current value, $currentVal$, with an expected value $expectedVal$. If the current value is equal to the expected value then a new value $newVal$ is set. In the fragment above $e.oldVersion$ refers to the *orec* initially seen by the entry; then if the *orec* known to the field is the same as the *orec* initially seen by the entry then no change has been made, and the new descriptor is installed in the orec. In either case the *orec* obtained by getOrec(e.field) is returned. If the CAS is successful then *acquire* returns $TRUE$ and continues until orecs for all entries have been acquired, or else the transaction is aborted. If all orecs are acquired then the commit can proceed. The failure cases for the acquire method are that the method call, $holds\_version\_number(seen)$, returns true - which is the case if an *orec* has only a version number and does not refer to a descriptor. This indicates that the field has been changed and there is a different version number seen than expected. This causes the *acquire* method to return $FALSE$ and the transaction is aborted. The remaining failure case is when the *orec* is being accessed by another transaction, and $BUSY$ is returned; again the transaction will be aborted.

There are a number of STM implementations such as that of the XSTM [4] project, which includes a Java version, JSTM; and another Java STM implementation, Multiverse [1].

8.5 Tooling

During our investigation of the transactional version of OCB we have translated code to Java manually, focussing on the practicalities of the approach rather than the tooling, so we have no Eclipse plug-ins for a Transactional-OCB meta-model, and we have no automatic translation tools. This is in part due to the fact that our proposed extensions to the RODIN tool, e.g. the sequential operator, would have to be introduced to make such a translation useful. The extension of the RODIN tool to accommodate this is beyond the scope of current work. To investigate and verify the lock management approach we used manual translation to Java code, and used specifications that had actions with parallel composition (instead of sequential). It would be possible to develop further a meta-model and translator plug-ins to automate this version but we leave that decision for the future.

There is some scope for optimisation of the lock acquisition process, even at this early stage of investigation. For example a simple exponential back-off approach can be used, where the thread goes to sleep for a successively longer duration after each failed attempt - before reattempting lock acquisition. An implementation of this can be seen in

Figure 8.12. An upper limit for waiting is supplied as a parameter, preliminary investigations indicate that a value of 5 provides a useful maximum sleep time (e^5ms = 148 ms). However the topic of optimising a system in such a way is beyond the scope of this piece of work, we merely wish to highlight that we are aware of the limitations of the approach.

```
protected void exponentialBackoff(int limit){
 if(backoff<limit){
   backoff++;
 }
 try {
   Thread.sleep((long) Math.exp(backoff));
 } catch (InterruptedException e) {
   e.printStackTrace();
 }
}
```

FIGURE 8.12: Exponential Back-off Java Implementation

8.6 Review of the Chapter

In this chapter we presented an extension to the approach where atomic actions are able to access more than one object. We note that for the target implementation a Software Transactional Memory approach may provide an efficient implementation. However we adopted a lock based approach in order to investigate its feasibility, and investigate the limitations that such an approach may give rise to. In this extension to original work we are able to remove some restrictions, the most significant of which is to enable more than one shared object to be referenced in an atomic action. We also introduce an atomic sequential construct which provides sequencing within an atomic statement. However the introduction of the atomic sequential construct does rely on an update to the existing Event-B approach to permit the use of a sequential operator in event actions. This would also give rise to a change in the proof obligation generator, which have to generate different proof obligations to accommodate the sequential construct. However we (as designers of the methodology) still need to be wary of the nested monitor problem, since this can still arise when using JLS3. The difference between JLS2 and JLS3 is that with JLS3 the programmer has the 'tools' to ensure the nested monitor problem does not occur; this is done by releasing all held locks when a condition causes a thread to wait. The benefit to Transactional-OCB users is that this should be taken care of by the tools.

We began the chapter with an overview of the Java Language Specification 3 locking and blocking features. The use of JLS3 allows us to remove many of the restrictions

when compared to Synchronized-OCB. The limitations imposed because of JLS2 were mainly due to the lack of control of monitor locking. We developed a new syntax, and introduce features such as direct access to objects using dot notation. We also introduced invariants that ensure well-definedness of each of the reference attributes. This ensures that null references are not accessed during either direct access or a procedure call. In Transactional-OCB we may have more than one method call per action; we change the conditional waiting approach so that conditions are declared (using the **when** clause) on entry to the labelled atomic clause, rather than in individual procedure definitions. This is because we may have more than one procedure call in an action. In this way it is easy to identify the entry conditions for the labelled action. We defined the translation of actions into Event-B, noting that the mapping at a higher level is the same as Synchronized-OCB; and we showed some simple examples to illustrate use of the new constructs. Then we discussed the issue of lock acquisition in the Java implementation, followed by the translation of the OCB notation to Java. We showed how conditional waiting is implemented in Java, illustrating its use in an example. We also discussed an alternative approach to locking using Software Transactional Memory. Finally we discussed tooling issues and implementation issues, noting that our locking strategy could be optimised.

Chapter 9

Conclusions and Future Work

In this thesis we have shown how to link an Event-B model to an object-oriented implementation by means of an intermediate specification using the OCB notation. The main contribution of the work is the introduction of means of specifying concurrent aspects of a development, rather than an attempt to incorporate object-oriented aspects into an Event-B development. An OCB specification incorporates the concurrent aspects of an implementation, allowing specification of process classes with interleaving, non-atomic operations. The non-atomic operations are comprised of a number of labelled atomic clauses. An atomic clause can be an atomic action, or an atomic procedure call. Data can be shared between processes using monitor classes with atomic procedure definitions. The transition between the Event-B development and OCB specification is greatly assisted by the use of diagrammatic representations of the link between abstract events and the implementation level constructs that implement and refine them. Since the diagrammatic representation describes sequences of atomic clauses in the abstraction and atomic clauses at the implementation level, it provides a relatively intuitive way to group related activities. These related activities, in the abstraction, can be implemented as related activities in the implementation; typically in the body of a single process. This is of course at the discretion of the developer who will be making implementation level design decisions.

An extension of the work allows access to multiple objects within a transactional, labelled clause. We call this approach Transactional-OCB to differentiate it from the simpler form, Synchronized-OCB. We believe that specification using OCB can ease the transition between formal modelling at an abstract level, and provision of a concurrent implementation. Reasoning about concurrency is simplified by providing a clear view of atomicity, which is achieved by abstracting away the locking details. We do not aim to provide formal verification of the link between the formal development and implementation, however we are confident that the semantic gap between the OCB and the implementation is sufficiently small for the relationship to be justified by inspection.

9.1 Review of Thesis

We began the thesis with an overview of our contribution and the structure of this thesis. We continued in Chapter 2 with an introduction to formal methods and discussed how object-oriented techniques and formal methods have influenced each other, and how they have been applied in the software development process. We went on to give details of the main issues that have influenced our work, including descriptions of the Event-B approach, UML-B, and B0. We followed this by discussing some programming problems associated with implementing concurrent Java programs, and then discussed some approaches for improving the dependability of Java programs.

Chapters 3 and 4 present the most significant contribution, it is here that we define the Synchronized-OCB syntax, present the rules for mapping Synchronized-OCB to Event-B and show an example translation to an Event-B model. We then look at producing a Java implementation from an OCB specification and define the translation rules for mapping OCB to Java. Initially the syntax and mapping from OCB to Event-B are defined using the Guarded Command Language where we introduce the notion of processes with non-atomic operations. These consist of labelled atomic clauses; the labels of the clauses map to program counter values, and these are used in guards to model the order of executions. A clause's action maps to an event action and a guard maps to an event guard. We introduced the notion of shared monitors; processes share monitors and access their data using atomic procedure calls. Mapping of procedure calls to Event-B results in in-line expansion of procedure bodies in the calling process. Input and return parameters were added, which involves substitution of formal parameters for actual parameters.

The modelling of object-oriented features is based on the approach that underlies the modelling of objects in UML-B [140]. Mapping of variables was discussed; each variable belongs to an OCB class and can be referred to in OCB clauses (in guards and actions). Due to the fact that we map an OCB specification to a model with instances, we require a translation function to map each occurrence of a variable in an OCB clause, to a variable associated with a specific instance in the corresponding Event-B clause. Finally we discuss the definition of OCB arrays, and their mapping to Event-B.

Following the presentation of the Event-B semantics underlying Synchronized-OCB we introduce syntactic sugar that provides a simple mapping to Java - for the branching, looping and guarding (conditional waiting) constructs. The syntactic sugar is a textual notation for use in specifications that is object-oriented in style, and is more appropriate for the specification of implementation related details than the guarded command syntax. We then presented an example translation from OCB to Event-B. The resulting Event-B model is intended to fit within the refinement approach to system development, since we can show that it refines some abstract model. However, the Event-B model of the implementation seems to be somewhat verbose when compared to the related Java

source code. This is due to the assumptions and hidden dependencies within a Java development, and in practice may lead to difficulty in establishing proof of refinement. We will therefore seek to rationalize the approach, which could be achieved by the development of some patterns and guidelines, and maybe a calculus. This will aid construction of OCB specifications from Event-B models. We have seen how the intricacies of the Java implementation can influence the design of the specification language, so mappings to other target implementations may further influence the OCB language. For instance, in future it will be interesting to investigate a mapping to the *RavenSpark* [2] subset of *SPARKAda*.

An example mapping to Java code was presented and the translation rules for the mapping from Synchronized-OCB to Java code was then defined. The OCB specification makes use of clearly defined atomic regions, which map to Java code with corresponding atomic regions. We are confident that the mapping will give rise to interference free execution, due to the restrictions we impose. We are also confident in the correctness of the correspondence between formal model and the implementation; however proof of this will be the subject of future work. In Chapter 5 we briefly outline the tooling issues associated with implementing our prototype OCB modelling and translation tools.

In the case study of Chapter 6 we showed how to specify and implement an object-oriented, concurrent system involving processes that read and write to shared buffers. We began with an abstract development and used diagrams, similar to Jackson Structure Diagrams of [86], to visualise the refinement and the OCB specification. We noted that behaviour such as looping and branching is more readily apparent in the Jackson Structure Diagram and is therefore an aid to visualising the development at the OCB specification level. We also noted that the size of the Event-B implementation refinement is large with respect to the size of the development, which leads us to conclude that models may need to be decomposed in future developments to make them manageable. At the OCB specification level we implemented a process class that could either read or write, depending on the parameter supplied, to a shared buffer; and the shared buffer was implemented as a monitor class. A main class can instantiate readers, writers and shared buffers, and in our implementation we created a single channel that was shared between two pairs of reading/writing processes. The translation resulted in a Java implementation that was executable and an Event-B model that was amenable to proof although the model itself was somewhat large and unwieldy. We note some of the difficulties experienced while trying to prove refinement of the abstract development; we propose a method of decomposing models to make them more manageable. The decomposition approach would also provide a way of reasoning about the model in a modular fashion, as well as providing the opportunity to reduce the abstraction gap between the abstract development and code.

The second case study of Chapter 7 describes the implementation of the User Application Layer API, and File System Layer API, of a flash file system based on the Intel Flash

File System Core Specification [82]. The API layers were implemented using an OCB specification, but the system described by the lower layer APIs were simulated due to the restriction to one procedure call per labelled atomic clause. The implementation of the case study using OCB specification made use of Jackson Structure type Diagrams which were again a useful aid to visualising the development. As in the previous case study the translator gave rise to Java code that was executable and an Event-B model that was amenable to proof. However in this case the lower level API layers of the Flash File System specification involve simulation. We believe that the Transactional-OCB extension will be of use here, to allow access to the lower layers of the hierarchical specification using dot-notation and getter methods. In Transactional-OCB extension objects may be nested, and procedure calls to those objects are permitted provided no state updates take place during the call. The approach is therefore to have procedure calls which are getter methods to be used in expressions; or updates can be made directly to nested objects using dot-notation after having acquired an object using a getter method.

We have developed prototype tool support for our approach, integrating with the RODIN Event-B tool. Details are presented in Chapter 5. The tool consists of several plug-ins which integrate with the Eclipse Platform [148]. The plug-ins contribute the OCB meta-model, and factory classes for instantiation of the meta-model; a tree-editor for construction of OCB specifications; an OCB text viewer to view the OCB specification as a text file; and the translators.

In our extension of the OCB approach of Chapter 8 we present Transactional-OCB. The transactional constructs allow direct access to multiple shared objects in an atomic region, as well as multiple procedure calls in an atomic region. Many of the restrictions that were imposed on Synchronized-OCB have been removed. We used the *java.utils.concurrent* packages for greater efficiency and flexibility, for instance techniques we overcome the nested monitor problem by controlling lock acquisition and release. The extended approach retains the concept of process classes of Synchronized-OCB, but instead of the $MonitorClass$ we have the $SharedClass$ construct. The procedures of the $SharedClass$ do not have waiting constructs, but the waiting construct now appears as the first clause of a labelled atomic action. In the extension the locks are acquired and released by lock managers that are created during the translation process.

The initial approach was to give synchronized-OCB similar an object-oriented look and feel in order to simplify the mapping to the Java target language. However an OCB specification is at a higher level of abstraction than a programming language such as Java and C. We abstract away details such as synchronization, or lock acquisition, and conditional waiting. In OCB, processes perform activities; monitors are shared between the processes, but are not active unless called by a process. OCB monitors provide an encapsulation mechanism giving the processes mutually exclusive access to the data. In our Java implementation processes are implemented using threads; and

the mutual exclusion mechanism is provided by appropriate use of data encapsulation, and the enforced use of synchronized methods. It is also the case that alternative target implementations should map to the OCB constructs relatively easily. For example, we can compare the process abstraction to Ada tasks [147], and the monitor abstraction to Ada protected objects. OCB procedures map easily Ada procedures and functions (functions may be used if no state updates are performed). Conditional waiting can be implemented using Ada entry barriers. The correspondence with C [93] is not quite so simple, but still feasible. Class data may be implemented using structs and the struct can be passed as an argument to functions performing updates to the class' data. This is similar in nature to Ada's idiom for implementation of Abstract Data Types. Thread implementation, and synchronization, can be done using the the features of a library such as POSIX [151]. The POSIX library for C provides a set of features including pthreads, mutex locks and conditional waiting.

9.2 Related Work

The motivation for this work was to explore the link between Event-B and object-oriented implementations; with the specific aim of discovering an approach facilitating concurrent implementations for formal developments undertaken using the Event-B method. To our knowledge no other work has been undertaken to facilitate this, so the foundations for our work are drawn from a number of areas. In this respect the nearest comparable work is that involving implementations for Classical-B [5] using the B0 implementation notation described in [43]. B0 is similar to a programming language, and consists only of concrete programming constructs that map to programming constructs in programming languages. B0 forms part of the Classical-B refinement chain, so the implementation level specification is shown to refine an abstract development. Translators are available to translate the implementation level specification to various target programming languages, which are described in [42] to executable code. Our OCB implementation notation exists at a similar level to B0 in the refinement chain, between the formal model and implementation; and similarly the constructs we chose for OCB have convenient mapping to constructs in our target domain - that of object-oriented programs. It should be noted though whilst B0 can be translated to the C++ programming language [146] there is no support for concurrent processing of threads; and the main factor for choosing Java in our work was its good support in this respect. Other target languages of the B0 translators are C [93], and *High Integrity Ada* (based on *SPARKAda* [2]). *SPARKAda* is a target programming language that we could consider translating to in future work since it incorporates proof of program consistency using a Design-by-Contract approach.

In the early stages of our investigations we considered using a combined CSP and B approach to specify the order in which the atomic actions may occur, and to specify

points at which the actions of processes may interleave. In such a development the specifications are combined so that the B operations synchronize with the corresponding CSP events with the same name. We considered this approach to be more complex than the approach we ultimately adopted, and introduced the sequence operator and labels for atomic clauses, to OCB. The sequence operator, together with the labels (mapped to program counters in Event-B), perform the same role as the CSP specification by imposing an ordering on the events. Additional considerations in making this choice, at the time, were the lack of tool support for combined Event-B and CSP, and the potential overlap with the work on JCSProB [162] that we discuss later. The combined CSP and B approach, $CSP \parallel B$, [131, 133] continues to be of interest to researchers.

Other work with CSP, related to our approach, is JCSP [157, 123] and JCSProB [162]. JCSP links the OCCAM [134] subset of the CSP process algebra and the Java programming language. The result is the ability to specify process behaviour in CSP, and translate to Java. The resulting Java is a message passing style implementation of communication between processes. This differs from the shared memory approach described in our work. JCSProB combines the CSP and classical-B formal methods, the *ProB* tool can be used to provide a unified approach for specification and model checking. The most obvious difference between the $JCSProB$ approach and OCB, once again, is that our work is aimed at the more recent Event-B approach. $JCSProB$ uses the CSP prefix operator to provide an ordering on the events that occur, with the operations of the B machine synchronizing with the CSP events. A CSP process may specify events that are not shared with other processes, this allows processes events to interleave - restricted only by the ordering imposed by the sequence operator (since the processes do not synchronize on common events). In OCB we can specify when interleaving may occur at certain points in non-atomic constructs, such as at the end of each while loop iteration. We also order the executions of labelled atomic clauses, using a sequential operator. The sequence operator is used in a process class' *run* operation to define an ordering of executions and to define points where other processes may interleave. In both $JCSProB$ and OCB the specifications are translated to threads which can run concurrently, and perform state updates.

Other work involving CSP is that of *Circus* [159], which is a combined approach using CSP and $Z - notation$ [144]. In a *Circus* specification the Z and CSP constructs are used to build a specification that is amenable to model checking using [160]. In this respect *Circus* has more in common with $JCSProB$ than OCB since it is a combined approach using model-checking technology, OCB is not a combined approach. *Circus* can be translated to Java as described in [65], making use of the JCSP library code.

The OCB approach we advocate has a strong object-oriented bias, which is in part due to our choice of target implementation. In addition though, the use of an object-oriented specification style in OCB has many benefits; such as the use of classes as templates for instantiating multiple objects and encapsulating data. The benefits of object-oriented

techniques have been applied to a number of formal approaches, one such approach involving object-oriented technology and Event-B, is that of UML-B. UML-B is a graphical front-end for Classical-B and Event-B and provides facilities to model a wide range of developments diagrammatically using class and statechart diagrams. We gained insight into the modelling of classes, objects, and ordering of events from UML-B; however there is no facility to translate the model to an implementation. The additional information contained in the $ProcessClass$ and $MonitorClass$ specifications of OCB, provide the necessary information to facilitate the translation to an object-oriented implementation. Due to the similarity of approaches it therefore possible that some UML-B developments will be intuitively refined by OCB specifications; in particular this would be quite apparent if a graphical front-end, similar to UML-B, were created for OCB.

There have been other approaches involving formal methods and object oriented technology. VDM++ is an object-oriented extension to VDM-SL formal specification language. Models can be described textually; or using a graphical interface using UML diagrams, in much the same way as UML-B does for B and Event-B. VDM++ can be used with the VDM++ Toolbox to generate C++ and Java code. VDM++ can be used to model and implement developments with concurrently executing processes, using threads. Conditional waiting can be specified using permission predicates, in a similar way to the OCB **when** construct. Another approach that is derived from VDM and additionally CSP is the RAISE Method [152]. The RAISE Method describes how formal development may be undertaken in a number of different ways. There are guidelines for applicative (functional) and imperative specification styles, for both sequential and concurrent systems. The approach covers the development activity from requirements specification through to translation, which is most relevant for this discussion. The RAISE method uses a specification notation called the RAISE Specification Language (RSL). A developer may initially make use of the language's high level specification constructs, that involve non-determinism, and may use a step-wise refinement approach to move towards implementation. RSL also contains the low level implementation constructs used at this level - unlike OCB in which we distinguish Event-B from OCB. The implementation level specification can then be translated into a traditional programming language, due to the similarity of the low level RSL constructs and traditional programming constructs. Following this the resulting code is compiled into executable, or interpreted code, in a similar way to the OCB approach. As with OCB the RAISE approach recognises the difficulty of formally linking the executable code and the related abstract specifications.

$Object - Z$ [136] is an object-oriented approach to development using Z. Unlike the work presented in the thesis, the Object-Z approach allows the use of inheritance and polymorphism in specifications. In Object-Z operations are blocked if the pre-condition is not true, which is similar to the OCB conditional waiting construct. A route to implementation is described using a translation to $PerfectDeveloper$ [55] in [145]. $PerfectDeveloper$'s approach is to use verified-Design By Contract, where verification

conditions are generated from a specification using constructs such as pre and postconditions, class and loop invariants, and assertions. The verification conditions must be shown to hold in order to show the specified contracts are satisfied by the implementation. They are generated for each method entry to show that the precondition holds, for each method exit to show that the postcondition holds, and wherever an assertion appears. $PerfectDeveloper$ provides automatic, and semi-automatic, translations to Java and C++ but appears not to support concurrent processing. In future work we will consider linking OCB to a verifiable implementation language such as $SPARKAda$, here concurrency may be facilitated using the $RavenSPARK$ version. Another approach would be to investigate the use of JML [102] annotations to specify a contract which we should ensure is satisfied.

9.3 Future Work

The work described in the thesis leaves open many opportunities for improvement, to both the existing approach and tool support, and for extensions to the approach with corresponding tool support. We now provide brief details of some of the work required to make the tool more usable, and implementations more efficient.

- Introduce the decomposition approach to provide smaller models at the implementation level.
- Improve the static checks that the tool performs.
- Improve integration with the RODIN tool, add the option to inject the Event-B output into an existing RODIN Project. Also improve the means of relating the abstract development to the OCB specification - at the moment relating the two is difficult, investigate patterns and tooling aids to enhance developer's productivity.
- Add a text-based editor (TEF based) with syntax checking and context highlighting and a graphical interface (GEF based).
- There are some smaller issues to be addressed, e.g. allow the use of guard predicates in a conditional statement, so $\neg(c)$ could contain a guard predicate c instead of restricting c to be an attribute of boolean type.
- In the Synchronized-OCB translators we need to replace approach for handling null references at the time of a procedure call with proof of its absence using suitable invariants.
- Model the Java integer range by adding a constant of interval type -2147483648 ... 2147483647 and use this to type OCB integer attributes.

- Optimise locking strategies, and investigate alternative approaches such as that based on Software Transactional Memory.

Other work can extend the functionality of OCB and Event-B,

- Extend to other target platforms, such as $SPARKAda$ or $C++$.
- Investigate the link between UML-B and OCB to make best use of features common to both.
- Investigate Re-use of OCB specifications using modularity and inheritance.
- Extend Event-B actions with the necessary operators, sequence, loop and branch.

Other issues are,

- Add non-blocking atomic procedure definitions to $ProcessClass$ definitions,
- The Transactional-OCB conditional waiting construct requires further work. A particular thread may be blocked when a guard fails, if the guard refers to more than one object the thread has to wait for changes to either shared object. An await/signal scheme needs to be implemented to facilitate this behaviour.

During our work we also identified the need for another kind of class. This is a class that performs a similar role to a $MonitorClass$ but is not shared. The class is simply part of the representation of some other object, so it does not need the same disciplined locking approach required by a shared class. The concept of such *representation objects* has been investigated in [127, 87].

Appendix A

Syntax for OCB

In the following, s is a symbol, $[s]$ denotes zero or one s is permissible, $s+$ denotes 1 or more s is permissible, and $s*$ denotes 0 or more is permissible.

$MainClass$ ::=
$CName$
$ProcessClass^*$
$MonitorClass^*$
Var^*
$NonAtomic$

$ProcessClass$ ::= $CName\ Var^+\ NonAtomic\ Constructor$

$MonitorClass$ ::= $CName\ Var^*\ Procedure^+\ Constructor$

$NonAtomic$ ::=
$NonAtomic$; $NonAtomic$
| $NonAtomic$ [] $NonAtomic$
| $do\ Atomic$ [; $NonAtomic$] od
| $Atomic$

$Atomic$ ::= $startLabel :< [Guard \rightarrow] Body >$

$Body$ ::=
$Action$
| $[v :=]\ m.pn(a_1, \ldots, a_k)$
| $v :=\ C.\mathbf{create}(a_1, \ldots, a_k)$

v ::=
$identifier$
| $identifier$‘[’ $IntegerLiteral$ ‘]’

Action ::=
Action ‖ *Action*
| $v := E$

Var ::=
$v \in Type$
$[v := E]$

$Type$ ::=
$CName$
| *Integer*
| *Boolean*
| $CName$‘[’ *IntegerLiteral* ‘]’
| *Integer*‘[’ *IntegerLiteral* ‘]’
| *Boolean*‘[’ *IntegerLiteral* ‘]’

E ::=
ArithmeticExpression
| *BooleanLiteral*

ArithmeticExpression ::=
IntegerLiteral
| v
| *ArithmeticExpression ArithBinOp ArithmeticExpression*
| *ArithUnOp ArithmeticExpression*

ArithBinOp ::=
$+ \mid - \mid * \mid / \mid mod \mid \wedge$

ArithUnOp ::=
$-$

Guard ::=
ArithmeticExpression GuardBinOp (*ArithmeticExpression* | *BooleanLiteral*)
| *Guard GuardBinOp Guard*
| *GuardUnOp Guard*

GuardBinOp ::=
$= \mid \neq \mid < \mid > \mid \leq \mid \geq \mid \wedge \mid \vee$

GuardUnOp ::=
$\neg$

APar ::= E

$Procedure ::= \ pdName \ LVar^* \ [Guard] \ Action$

$LVar ::= \ v \in Type$

$Constructor ::= \ LVar^* \ Action \ Type$

Appendix B

OCB Syntax Extension

In the following, s is a symbol, $[s]$ denotes zero or one s is permissible, $s+$ denotes 1 or more s is permissible, and $s*$ denotes 0 or more is permissible. Symbols beginning with an upper case character are non-terminals and are terminals otherwise.

$MainClass$ ::=
$CName$
$ProcessClass^*$
$PassiveClass^*$
Var^*
$NonAtomic$

$ProcessClass$ ::= $CName\ Var^+\ NonAtomic\ Constructor$

$PassiveClass$ ::= $CName\ Var^*\ Procedure^+\ Constructor$

$NonAtomic$ ::=
$NonAtomic$; $NonAtomic$
| $NonAtomic$ [] $NonAtomic$
| $do\ Atomic$ [; $NonAtomic$] od
| $Atomic$

$Atomic$ ::= $startLabel :< [Guard\rightarrow]\ Action >$

$Action$ ::=
$Action$; $Action$
| **if**($Guard$) **then** $Action$ **endif**
[**else if**($Guard$) **then** $Action$ **endelseif**]
[**else** $Action$ **endelse**]
| **while**($Guard$) **then** $Action$ **endwhile**
| $identifier := E$

$| [identifier :=] identifier.pn(a_1, \ldots, a_k)$
$| identifier := C.\mathbf{create}(a_1, \ldots, a_k)$

$identifier ::=$
$identifier$
$| identifier$‘[’ $IntegerLiteral$ ‘]’
$| identifier$ ‘ . ’ $identifier$

$Var ::=$
$identifier \in Type$
$[identifier := E]$

$Type ::=$
$CName$
$| Integer$
$| Boolean$
$| CName$‘[’ $IntegerLiteral$ ‘]’
$| Integer$‘[’ $IntegerLiteral$ ‘]’
$| Boolean$‘[’ $IntegerLiteral$ ‘]’

$E ::=$
$ArithmeticExpression$
$| BooleanLiteral$

$ArithmeticExpression ::=$
$IntegerLiteral$
$| identifier$
$| identifier.pn(a_1, \ldots, a_k)$
$| ArithmeticExpression\ ArithBinOp\ ArithmeticExpression$
$| ArithUnOp\ ArithmeticExpression$

$ArithBinOp ::=$
$+ | - | * | / | mod | \wedge$

$ArithUnOp ::=$
$-$

$Guard ::=$
$ArithmeticExpression\ GuardBinOp\ (ArithmeticExpression | BooleanLiteral)$
$| Guard\ GuardBinOp\ Guard$
$|\ GuardUnOp\ Guard$

$GuardBinOp ::=$
$= | \neq | < | > | \leq | \geq | \wedge | \vee$

$GuardUnOp ::=$
$\quad \neg$

$APar ::= E$

$Procedure ::= pdName\ LVar^*\ [Guard]\ ProcAction$

$LVar ::= identifier \in Type$

$Constructor ::= LVar^*\ Action\ Type$

$ProcAction ::=$
$\quad ProcAction\ ;\ ProcAction$
$\quad |\ \textbf{if}(Guard)\ \textbf{then}\ ProcAction\ \textbf{endif}$
$\quad |\ \textbf{while}(Guard)\ \textbf{do}\ ProcAction\ \textbf{endwhile}$
$\quad |\ identifier := E$
$\quad |\ [identifier :=]\ InternalProcCall$

$InternalProcCall ::= pdName\ APar^*$

Appendix C

Case Study 1 - OCB and Event-B Models, and Code

C.1 OCB Channel Specification

```
MonitorClass Channel{
 // Attributes
 Integer[50] buff, Integer rLoc, Integer wLoc,
 Integer rPID, Integer wPID, Integer writeSize

 // The Constructor
 Procedure create(){
  rLoc:= 0 || wLoc:= 0 || rPID:= -1 ||
  wPID:= -1 || writeSize:= -1
 }

 // 'Refines' WritePacket - in a call from clause p4
 Procedure add(Integer val){
  when(wLoc - rLoc <= 5){
        buff[wLoc]:= val || wLoc:= wLoc + 1}
 }

 // The value is stored in a temporary buffer in a
 // call from clause p8 - implementing ReadPacket
 //as part of the reading activity.
 Procedure remove(){
  when(wLoc - rLoc > 0){
        return:= buff[rLoc] || rLoc := rLoc+1 }
```

```
 }: Integer

 // Called in p1_else clause - refines StartRead.
 // Set the channel for reading, by the process
 // with identifier pid.
 // Block if it is already owned or has nothing to read.
 Procedure getRChan(Integer pid){
  when(rPID=-1 & writeSize>0){rPID:= pid}
 }

 // Called in p11 clause - refines EndRead.
 // Free the channel for reading.
 Procedure freeRChan(){
  rPID:= -1 || writeSize:= -1
 }

 // Called in p1 clause - implementing StartWrite.
 // Set the channel for writing writesze bytes, by
 // the process pid.
 // Block if the channel is already owned for writing or
 // has bytes still to write.
 Procedure getWChan(Integer pid,Integer writeSze){
  when(wPID=-1 & writeSize<=0){
        wPID:= pid || writeSize:= writeSze}
 }

 // Called in p6 clause - refines EndWrite.
 // Free the channel for writing.
 Procedure freeWChan(){ wPID:= -1 }

 // Return the number of bytes to write.
 Procedure getWriteSize(){ return:= writeSize }: Integer
}
```

C.2 OCB ProcessClass Specification

```
ProcessClass Proc{
 // Attributes
 Buffer buff, Boolean isWriter, Channel c, Integer id,
```

```
 Integer tmpBuffSz, Integer tmpDat
 // The constructor procedure
 Procedure create(Integer pid, Buffer bff,
                      Boolean isWritr, Channel ch){
  id:=pid || buff:=bff || isWriter:=isWritr || c:= ch ||
  tmpBuffSz:=-1 || tmpDat:=-1
 }
 // The process behaviour
 Operation run(){
  p1: if(isWriter=TRUE) then
       tmpBuffSz:=buff.getSize() andthen
       p2: c.getWChan(id, tmpBuffSz);   // Clause p2 refines StartWrite
       p3: while(tmpBuffSz>0) do tmpDat:=buff.remove() andthen
            p4: c.add(tmpDat);  // Clause p4 refines WritePacket
            p5: tmpBuffSz:=tmpBuffSz-1 endwhile ;
       p6: c.freeWChan() endif // Clause p6 refines EndWrite
  else c.getRChan(id) andthen // Clause p1_else refines StartRead
       p7: tmpBuffSz:=c.getWriteSize();
       p8: while(tmpBuffSz>0) do tmpDat:=c.remove() andthen
            p9: buff.add(tmpDat); // Clause p9 refines ReadPacket
            p10: tmpBuffSz:=tmpBuffSz-1 endwhile ;
       p11: c.freeRChan() endelse // Clause p11 refines EndRead
 }
}
```

C.3 OCB MainClass Specification

```
MainClass CommBuffer{
 // Attributes
 Buffer rBuff1, Buffer rBuff2, Buffer wBuff1, Buffer wBuff2,
 Channel chan, Proc wProc1, Proc wProc2, Proc rProc1,
 Proc rProc2, Integer v:=5

 // Program entry point.
 Operation main(){
  m1: rBuff1:=Buffer.create();
  m2: rBuff2:=Buffer.create();
  m3: wBuff1:=Buffer.create();
  m4: wBuff2:=Buffer.create();
  m5: chan:=Channel.create();
```

```
  m6: while(v<8) do wBuff1.add(v) andthen
       m7: wBuff2.add(v);
       m8: v:=v+1 endwhile ;
  m9: wProc1:=Proc.create(1,wBuff1,TRUE,chan);
  m10: wProc2:=Proc.create(2,wBuff2,TRUE,chan);
  m11: rProc1:=Proc.create(3,rBuff1,FALSE,chan);
  m12: rProc2:=Proc.create(4,rBuff2,FALSE,chan)
 }
}
```

C.4 OCB Buffer Specification

```
MonitorClass Buffer{
 // Attributes
 Integer[50] buff, Integer rLoc, Integer wLoc,

 Procedure create(){
  rLoc:=0 || wLoc:=0
 }

 // Add val to the channel buffer wLoc
 Procedure add(Integer val){
  when(wLoc - rLoc <= 5){
        buff[wLoc]:= val || wLoc:= wLoc + 1}
 }

 // remove, and return, the value at the rLoc of the channel buffer
 Procedure remove(){
  when(wLoc - rLoc > 0){
        return:= buff[rLoc] || rLoc := rLoc+1 }
 }: Integer
}
```

C.5 Channel Java Code

```
public class Channel {
 private int[] buff = new int[50];
 private int wLoc;private int rLoc;
 private int rPID;private int wPID;private int writeSize;
```

```
public Channel() {
  wLoc = 0; rLoc = 0; rPID = -1;
  wPID = -1; writeSize = -1;
}

public synchronized void add(int val) {
  try {
    while (!(wLoc - rLoc <= 5)) {
      wait();
  }} catch (InterruptedException e) { e.printStackTrace(); }
  buff[wLoc] = val;
  wLoc = wLoc + 1;
  notifyAll();
}

public synchronized int remove() {
  int initial_rLoc = rLoc;
  try {
    while (!(wLoc - rLoc > 0 )) {
      wait();
      initial_rLoc = rLoc;
    } catch (InterruptedException e) { e.printStackTrace();}
  rLoc = rLoc + 1;
  notifyAll();
  return buff[initial_rLoc];
}

public synchronized void getRChan(int pid) {
  try {
  while (!(rPID == -1 && writeSize > 0)) {
    wait();
  }} catch (InterruptedException e) { e.printStackTrace(); }
  rPID = pid;
  notifyAll();
}

public synchronized void freeRChan() {
  rPID = -1;
  writeSize = -1;
  notifyAll();
}
```

```
 public synchronized void getWChan(int pid, int writeSze) {
  try {
   while (!(wPID == -1 && writeSize <= 0)) {
     wait();
   }} catch (InterruptedException e) { e.printStackTrace(); }
  wPID = pid;
  writeSize = writeSze;
  notifyAll();
  }

 public synchronized void freeWChan() {
  wPID = -1;
  notifyAll();
 }

 public synchronized int getWriteSize() {
  return writeSize;
 }
}
```

C.6 The Proc Class Java Code

```
public class Proc implements Runnable {

 private Buffer buff = null; private boolean isWriter;
 private  Channel c = null; private int id;
 private int tmpBuffSz; private int tmpDat;

 public Proc(int pid, Buffer bff, boolean isWritr, Channel ch) {
  id = pid; buff = bff; isWriter = isWritr; c = ch;
  tmpBuffSz = -1; tmpDat = -1;
 }

 public void run() {
  if (isWriter == true) {
   tmpBuffSz = buff.getSize(); // p1
   c.getWChan(id, tmpBuffSz); // p2
   while (tmpBuffSz > 0) {
     tmpDat = buff.remove(); // p3
```

```
      c.add(tmpDat); // p4
      tmpBuffSz = tmpBuffSz - 1; // p5
    }
    c.freeWChan(); // p6
  }else {
    c.getRChan(id);
    tmpBuffSz = c.getWriteSize(); // p7
    while (tmpBuffSz > 0) {
      tmpDat = c.remove(); // p8
      buff.add(tmpDat); // p9
      tmpBuffSz = tmpBuffSz - 1; // p10
    }
    c.freeRChan(); // p11
  }
 }
}
```

C.7 The CommBuffer Java Code

```
public class CommBuffer {
 private static Buffer rBuff1 = null;
 private static Buffer rBuff2 = null;
 private static Buffer wBuff1 = null;
 private static Buffer wBuff2 = null;
 private static Channel chan = null;
 private static Proc wProc1 = null;
 private static Proc wProc2 = null;
 private static Proc rProc1 = null;
 private static Proc rProc2 = null;
 private static int v = 5;

 public static void main(String[] args) {
  rBuff1 = new Buffer(); // m1
  rBuff2 = new Buffer(); // m2
  wBuff1 = new Buffer(); // m3
  wBuff2 = new Buffer(); // m4
  chan = new Channel(); // m5
  while (v < 8) {
   wBuff1.add(v); // m6
   wBuff2.add(v); // m7
```

```
    v = v + 1; // m8
  }
  wProc1 = new Proc(1, wBuff1, true, chan);
  new Thread(wProc1).start(); /* m9 */
  wProc2 = new Proc(2, wBuff2, true, chan);
  new Thread(wProc2).start(); /* m10 */
  rProc1 = new Proc(3, rBuff1, false, chan);
  new Thread(rProc1).start(); /* m11 */
  rProc2 = new Proc(4, rBuff2, false, chan);
  new Thread(rProc2).start(); /* m12 */
 }
}
```

C.8 Event-B Model of Shared Channel

MACHINE CommBufferSimple

REFINES m4

SEES CommBufferSimple_CTX

VARIABLES

Buffer, Buffer_buff, Buffer_wLoc, Buffer_rLoc, Channel, Channel_buff, Channel_wLoc, Channel_rLoc,Channel_rPID, Channel_wPID, Channel_writeSize, Proc, Proc_state, Proc_buff, Proc_isWriter, Proc_c, Proc_id, Proc_tmpBuffSz, Proc_tmpDat, CommBufferSimple, CommBufferSimple_rBuff1, CommBufferSimple_rBuff2, CommBufferSimple_wBuff1, CommBufferSimple_wBuff2, CommBufferSimple_chan, CommBufferSimple_wProc1, CommBufferSimple_wProc2, CommBufferSimple_rProc1, CommBufferSimple_rProc2, CommBufferSimple_v, CommBufferSimple_state

INVARIANTS

Buffer$\in \mathbb{P}$ (Buffer_Set)
Buffer_buff$\in$Buffer $\rightarrow$ (0 .. 49 $\rightarrow$ $\mathbb{Z}$)
Buffer_wLoc$\in$Buffer $\rightarrow$ $\mathbb{Z}$
Buffer_rLoc$\in$Buffer $\rightarrow$ $\mathbb{Z}$
Channel$\in \mathbb{P}$ (Channel_Set)
Channel_buff$\in$Channel $\rightarrow$ (0 .. 49 $\rightarrow$ $\mathbb{Z}$)
Channel_wLoc$\in$Channel $\rightarrow$ $\mathbb{Z}$
Channel_rLoc$\in$Channel $\rightarrow$ $\mathbb{Z}$

```
Channel_rPID∈Channel → ℤ
Channel_wPID∈Channel → ℤ
Channel_writeSize∈Channel → ℤ
Proc∈ ℙ (Proc_Set)
Proc_state ∈ Proc ⇸ Proc_states
Proc_buff∈Proc ⇸ Buffer
Proc_isWriter∈Proc → BOOL
Proc_c∈Proc ⇸ Channel
Proc_id∈Proc → ℤ
Proc_tmpBuffSz∈Proc → ℤ
Proc_tmpDat∈Proc → ℤ
CommBufferSimple∈CommBufferSimple_Set
CommBufferSimple∈ ℙ(CommBufferSimple_Set)
CommBufferSimple_rBuff1∈CommBufferSimple ⇸ Buffer
CommBufferSimple_rBuff2∈CommBufferSimple ⇸ Buffer
CommBufferSimple_wBuff1∈CommBufferSimple ⇸ Buffer
CommBufferSimple_wBuff2∈CommBufferSimple ⇸ Buffer
CommBufferSimple_chan∈CommBufferSimple ⇸ Channel
CommBufferSimple_wProc1∈CommBufferSimple ⇸ Proc
CommBufferSimple_wProc2∈CommBufferSimple ⇸ Proc
CommBufferSimple_rProc1∈CommBufferSimple ⇸ Proc
CommBufferSimple_rProc2∈CommBufferSimple ⇸ Proc
CommBufferSimple_v∈CommBufferSimple → ℤ
CommBufferSimple_state ∈ CommBufferSimple ⇸ CommBufferSimple_states
Channel = chan
Proc = proc
∀ p,q · p∈Proc ∧
     q∈Proc ∧
     p ≠ q
      ⇒
     Proc_id(p) ≠ Proc_id(q) // all Proc_ids must be different
∀ self · self ∈ Proc ∧
     self ∈ dom(Proc_id) ⇒
     Proc_id(self) ≥ 0 // Process IDs are Natural numbers
∀ self · self ∈ Proc ∧
     self ∈ dom(Proc_buff) ⇒
     Buffer_rLoc(Proc_buff(self)) ∈ 0 .. 49 // Bound on Buffer rLoc
∀ self · self ∈ Proc ∧
     self ∈ dom(Proc_buff) ⇒
     Buffer_wLoc(Proc_buff(self)) ∈ 0 .. 49 // Bound on Buffer wLoc
∀ self · self ∈ Proc ∧
```

```
      self ∈ dom(Proc_c)  ⇒
      Channel_wLoc(Proc_c(self)) ∈ 0  ..  49 // Bound on Channel wLoc
∀  self  ·  self ∈ Proc  ∧
      self ∈ dom(Proc_c)  ⇒
      Channel_rLoc(Proc_c(self))∈0  ..  49 // Bound on Channel rLoc
∀ self  ·  self∈ CommBufferSimple  ∧
      self∈dom(CommBufferSimple_wBuff1)
       ⇒
      Buffer_rLoc(CommBufferSimple_wBuff1(self)) ∈ 0 .. 49
∀ self  ·  self∈ CommBufferSimple  ∧
      self∈dom(CommBufferSimple_wBuff1)
       ⇒
      Buffer_wLoc(CommBufferSimple_wBuff1(self)) ∈ 0 .. 49
∀ self  ·  self∈ CommBufferSimple  ∧
      self∈dom(CommBufferSimple_wBuff2)
       ⇒
      Buffer_rLoc(CommBufferSimple_wBuff2(self)) ∈ 0 .. 49
∀ self  ·  self∈ CommBufferSimple  ∧
      self∈dom(CommBufferSimple_wBuff2)
       ⇒
      Buffer_wLoc(CommBufferSimple_wBuff2(self)) ∈ 0 .. 49
∀ self  ·  self∈ CommBufferSimple  ∧
      self∈dom(CommBufferSimple_chan)
       ⇒
      Channel_wLoc(CommBufferSimple_chan(self)) ∈ 0 .. 49
∀ self  ·  self∈ CommBufferSimple  ∧
      self∈dom(CommBufferSimple_chan)
       ⇒
      Channel_rLoc(CommBufferSimple_chan(self)) ∈ 0 .. 49
∀ self  ·  self∈ CommBufferSimple  ∧
      self∈dom(CommBufferSimple_rBuff1)
       ⇒
      Buffer_rLoc(CommBufferSimple_rBuff1(self)) ∈ 0 .. 49
∀ self  ·  self∈ CommBufferSimple  ∧
      self∈dom(CommBufferSimple_rBuff1)
       ⇒
      Buffer_wLoc(CommBufferSimple_rBuff1(self)) ∈ 0 .. 49
∀ self  ·  self∈ CommBufferSimple  ∧
      self∈dom(CommBufferSimple_rBuff2)
       ⇒
      Buffer_rLoc(CommBufferSimple_rBuff2(self)) ∈ 0 .. 49
```

```
∀ self · self∈ CommBufferSimple ∧
    self∈dom(CommBufferSimple_rBuff2)
     ⇒
    Buffer_wLoc(CommBufferSimple_rBuff2(self)) ∈ 0 .. 49
∀ self · self ∈ Proc ∧
    self ∈ dom(Proc_c) ∧
    Channel_wPID(Proc_c(self)) = -1
     ⇒
    self ∉ dom(writing) // Channel wPID is unset means that
                        //the process is not in the writing set
∀ self · self ∈ Proc ∧
    self ∈ dom(Proc_c) ∧
    Channel_wPID(Proc_c(self)) = -1
     ⇒
    Proc_c(self) ∉ ran(writing) // Channel wPID is unset means that
                        //the Channel is not in the writing set Channel
∀ self · self ∈ Proc ∧
    self ∈ dom(Proc_c) ∧
    Channel_rPID(Proc_c(self)) = -1
     ⇒
    Proc_c(self) ∉ ran(reading) // Channel rPID unset means that
                        // the Channel is not in the reading set
∀ self · self ∈ Proc ∧
    self ∈ dom(Proc_c) ∧
    Channel_rPID(Proc_c(self)) = -1
     ⇒
    self ∉ dom(reading) // Channel rPID is unset means
                        // that the process is not in the reading set
∀ self · self ∈ Proc ∧
    self ∈ dom(Proc_state) ∧
    Proc_state(self) = p2
     ⇒
    self ∉ dom(reading) //  the process counter is at p2 means
                        // that the process is not reading
∀ self · self ∈ Proc ∧
    self ∈ dom(Proc_state) ∧
    Proc_state(self) = p1
     ⇒
    self ∉ dom(reading)
    // If Counter = p1 then the process is not reading
∀ self · self ∈ Proc ∧
```

```
      self ∈ dom(Proc_c)
       ⇒
      dom(data2(Proc_c(self))) = 0 .. 49
∀ self · self ∈ CommBufferSimple ∧
      self ∈ dom(CommBufferSimple_chan) ∧
      CommBufferSimple_chan(self) ∈ dom(data2)
       ⇒
      dom(data2(CommBufferSimple_chan(self))) = 0 .. 49
∀ self · self ∈ Proc ∧
      self ∈ dom(Proc_state) ∧
      Proc_state(self) = p1 ∧
      self∈dom(Proc_buff) ∧
      self∈dom(buff2) ∧
      Buffer_rLoc(Proc_buff(self)) > 0
       ⇒
      buff2(self) ≠ ∅ // If a buffer's read location > 0
                      // when pc = p1 then the buff2 is not empty
∀ self · self ∈ Proc ∧
      self ∈ dom(Proc_state) ∧
      Proc_state(self) = p2 ∧
      self∈dom(Proc_buff) ∧
      self∈dom(buff2) ∧
      Buffer_rLoc(Proc_buff(self)) > 0
       ⇒
      buff2(self) ≠ ∅ // the tmpBuffSz > 0 implies that
                      //buff2 is non-empty when pc =p2
```

THEOREM

```
( λ i · i∈0 .. 49 0) ∈ (0 .. 49 → ℤ )
```

EVENTS

```
    INITIALISATION ≙
    THEN
      Buffer Buffer := ∅
      Buffer_buff Buffer_buff := ∅
      Buffer_wLoc Buffer_wLoc := ∅
      Buffer_rLoc Buffer_rLoc := ∅
      Channel Channel := ∅
      Channel_buff Channel_buff := ∅
      Channel_wLoc Channel_wLoc := ∅
      Channel_rLoc Channel_rLoc := ∅
      Channel_rPID Channel_rPID := ∅
```

```
        Channel_wPID Channel_wPID := ∅
        Channel_writeSize Channel_writeSize := ∅
        Proc Proc := ∅
        Proc_state Proc_state := ∅
        Proc_buff Proc_buff := ∅
        Proc_isWriter Proc_isWriter := ∅
        Proc_c Proc_c := ∅
        Proc_id Proc_id := ∅
        Proc_tmpBuffSz Proc_tmpBuffSz := ∅
        Proc_tmpDat Proc_tmpDat := ∅
        CommBufferSimple CommBufferSimple := ∅
        CommBufferSimple_rBuff1 := ∅
        CommBufferSimple_rBuff2 := ∅
        CommBufferSimple_wBuff1 := ∅
        CommBufferSimple_wBuff2 := ∅
        CommBufferSimple_chan := ∅
        CommBufferSimple_wProc1 := ∅
        CommBufferSimple_wProc2 := ∅
        CommBufferSimple_rProc1 := ∅
        CommBufferSimple_rProc2 := ∅
        CommBufferSimple_v := ∅
        CommBufferSimple_state := ∅
    END

Proc_p1 ≙
      ANY self, target
      WHERE
       self ∈ Proc
       self ∈ dom(Proc_state)
       Proc_state(self) = p1
       Proc_isWriter(self) = TRUE
       self ∈ dom( Proc_buff )
       target = Proc_buff(self)
      THEN
       Proc_tmpBuffSz ( self ) :=
            Buffer_wLoc ( target ) - Buffer_rLoc ( target )
       Proc_state(self) := p2
    END

Proc_p1_isNull ≙
      ANY self
```

```
      WHERE
       self ∈ Proc
       self ∈ dom(Proc_state)
       Proc_state(self) = p1
       Proc_isWriter(self) = TRUE
        ¬ ( self ∈ dom( Proc_buff ) )
      THEN
       Proc_state(self) := terminatedProc
    END

Proc_p2 ≙
  REFINES StartWrite
      ANY self, target
      WHERE
       self ∈ Proc
       self ∈ dom(Proc_state)
       Proc_state(self) = p2
       self ∈ dom( Proc_c )
       target = Proc_c(self)
       Channel_wPID ( target ) = - 1
       Channel_writeSize ( target ) ≤ 0
       Proc_tmpBuffSz ( self ) ≥ 0
      WITH
        p = self
        c = Proc_c(self)
      THEN
       Channel_wPID ( target ) := Proc_id ( self )
       Channel_writeSize ( target ) := Proc_tmpBuffSz ( self )
       Proc_state(self) := p3
    END

  Proc_p2_isNull ≙
      ANY self
      WHERE
        self ∈ Proc
        self ∈ dom(Proc_state)
        Proc_state(self) = p2
         ¬ ( self ∈ dom( Proc_c ) )
      THEN
        Proc_state(self) := terminatedProc
    END
```

```
Proc_while_p3 ≙
    ANY self, target
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p3
      Proc_tmpBuffSz(self) > 0
      self ∈ dom( Proc_buff )
      target = Proc_buff(self)
      Buffer_wLoc ( target ) - Buffer_rLoc ( target ) > 0
    THEN
      Proc_tmpDat ( self ) :=
           Buffer_buff ( target ) ( Buffer_rLoc ( target ) )
      Buffer_rLoc ( target ) := Buffer_rLoc ( target ) + 1
      Proc_state(self) := p4
  END

Proc_while_p3_isNull ≙
    ANY self
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p3
      Proc_tmpBuffSz(self) > 0
       ¬ ( self ∈ dom( Proc_buff ) )
    THEN
      Proc_state(self) := terminatedProc
  END

Proc_p4 ≙
REFINES WritePacket
    ANY self, target
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p4
      self ∈ dom( Proc_c )
      target = Proc_c(self)
      Channel_wLoc ( target ) - Channel_rLoc ( target ) ≤ 5
      Channel_wLoc ( target ) < 49
```

```
    WITH
      p = self
      c = target
      k = Channel_wLoc( target )
      d = Proc_tmpDat( self )
    THEN
      Channel_buff ( target ) := 
         Channel_buff ( target ) ⩤
                 {Channel_wLoc ( target ) ↦ Proc_tmpDat ( self )}
      Channel_wLoc ( target ) := Channel_wLoc ( target ) + 1
      Proc_state(self) := p5
END

 Proc_p4_isNull ≙
    ANY self
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p4
       ¬ ( self ∈ dom( Proc_c ) )
    THEN
      Proc_state(self) := terminatedProc
  END

 Proc_p5 ≙
    ANY self
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p5
    THEN
      Proc_tmpBuffSz ( self ) := Proc_tmpBuffSz ( self ) - 1
      Proc_state(self) := p3
  END

 Proc_while_p3_false ≙
    ANY self
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p3
```

```
      ¬ ( Proc_tmpBuffSz(self) > 0  )
    THEN
      Proc_state(self)  :=  p6
  END

Proc_p6  ≙
    REFINES EndWrite
    ANY self, target
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p6
      self ∈ dom( Proc_c )
      target = Proc_c(self)
    WITH
      p = self
      c = Proc_c(self)
    THEN
      Channel_wPID ( target )  :=  - 1
      Proc_state(self)  :=  terminatedProc
  END

Proc_p6_isNull  ≙
    ANY self
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p6
       ¬ ( self ∈ dom( Proc_c ) )
    THEN
      Proc_state(self)  :=  terminatedProc
  END

Proc_p1_else  ≙
    REFINES StartRead
    ANY self, target
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p1
       ¬ (Proc_isWriter(self) = TRUE )
```

```
      self ∈ dom( Proc_c )
      target = Proc_c(self)
      Channel_rPID ( target ) = - 1
      Channel_writeSize ( target ) > 0
    WITH
      p = self
      c = target
    THEN
      Channel_rPID ( target ) := Proc_id ( self )
      Proc_state(self) := p7
  END

Proc_p1_else_isNull ≙
    ANY self
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p1
       ¬ (Proc_isWriter(self) = TRUE )
       ¬ ( self ∈ dom( Proc_c ) )
    THEN
      Proc_state(self) := terminatedProc
  END

Proc_p7 ≙
    ANY self, target
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p7
      self ∈ dom( Proc_c )
      target = Proc_c(self)
    THEN
      Proc_tmpBuffSz ( self ) := Channel_writeSize ( target )
      Proc_state(self) := p8
  END

Proc_p7_isNull ≙
    ANY self
    WHERE
      self ∈ Proc
```

```
      self ∈ dom(Proc_state)
      Proc_state(self) = p7
       ¬ ( self ∈ dom( Proc_c ) )
    THEN
      Proc_state(self)  :=  terminatedProc
  END

Proc_while_p8  ≙
    ANY self, target
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p8
      Proc_tmpBuffSz(self) > 0
      self ∈ dom( Proc_c )
      target = Proc_c(self)
      Channel_wLoc ( target ) - Channel_rLoc ( target ) > 0
      Channel_rLoc ( target ) < 50
    THEN
      Proc_tmpDat ( self )  :=
                Channel_buff ( target ) ( Channel_rLoc ( target ) )
      Channel_rLoc ( target )  :=  Channel_rLoc ( target ) + 1
      Proc_state(self)  :=  p9
  END

Proc_while_p8_isNull  ≙
    ANY self
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p8
      Proc_tmpBuffSz(self) > 0
       ¬ ( self ∈ dom( Proc_c ) )
    THEN
      Proc_state(self)  :=  terminatedProc
  END

Proc_p9  ≙
    REFINES ReadPacket
    ANY self, target
    WHERE
```

```
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p9
      self ∈ dom( Proc_buff )
      target = Proc_buff(self)
      Buffer_wLoc ( target ) < 49
      self ∈ dom(Proc_c)
    WITH
      p = self
      c = Proc_c(self)
      k = Buffer_wLoc( target )
    THEN
      Buffer_buff ( target ) :=
             Buffer_buff ( target ) ⩤
                    {Buffer_wLoc ( target ) ↦ Proc_tmpDat ( self )}
      Buffer_wLoc ( target ) := Buffer_wLoc ( target ) + 1
      Proc_state(self) := p10
  END

Proc_p9_isNull ≙
    ANY self
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p9
       ¬ ( self ∈ dom( Proc_buff ) )
    THEN
      Proc_state(self) := terminatedProc
  END

Proc_p10 ≙
    ANY self
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p10
    THEN
      Proc_tmpBuffSz ( self ) := Proc_tmpBuffSz ( self ) - 1
      Proc_state(self) := p8
  END
```

```
Proc_while_p8_false  ≙
    ANY self
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p8
       ¬ ( Proc_tmpBuffSz(self) > 0  )
    THEN
      Proc_state(self)  :=  p11
  END

Proc_p11 ≙
   REFINES EndRead
    ANY self, target
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p11
      self ∈ dom( Proc_c )
      target = Proc_c(self)
    WITH
      p = self
      c = target
    THEN
      Channel_rPID ( target )  :=  - 1
      Channel_writeSize ( target )  :=  - 1
      Channel_wLoc ( target )  :=  0
      Channel_rLoc ( target )  :=  0
      Proc_state(self)  :=  terminatedProc
  END

Proc_p11_isNull  ≙
    ANY self
    WHERE
      self ∈ Proc
      self ∈ dom(Proc_state)
      Proc_state(self) = p11
       ¬ ( self ∈ dom( Proc_c ) )
    THEN
     Proc_state(self)  :=  terminatedProc
  END
```

```
loadCommBufferSimple ≙
    ANY self
    WHERE
      self self ∈ CommBufferSimple_Set \ CommBufferSimple
    THEN
      CommBufferSimple := CommBufferSimple ∪ {self}
      CommBufferSimple_state(self) := m1
      CommBufferSimple_v(self) := 5
  END

CommBufferSimple_m1 ≙
    ANY self, new
    WHERE
     self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      new ∈ Buffer_Set \ Buffer
      CommBufferSimple_state(self) = m1
    THEN
      Buffer_buff(new) := λ i · i∈0 .. 490
      Buffer_wLoc ( new ) := 0
      Buffer_rLoc ( new ) := 0
      Buffer := Buffer ∪ {new}
     CommBufferSimple_rBuff1(self) := new
     CommBufferSimple_state(self) := m2
  END

CommBufferSimple_m2 ≙
    ANY self, new
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      new ∈ Buffer_Set \ Buffer
      CommBufferSimple_state(self) = m2
    THEN
      Buffer_buff(new) := λ i · i∈0 .. 490
      Buffer_wLoc ( new ) := 0
      Buffer_rLoc ( new ) := 0
      Buffer := Buffer ∪ {new}
      CommBufferSimple_rBuff2(self) := new
      CommBufferSimple_state(self) := m3
```

```
  END

CommBufferSimple_m3 ≙
    ANY self, new
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      new ∈ Buffer_Set ∖ Buffer
      CommBufferSimple_state(self) = m3
    THEN
      Buffer_buff(new) := λ i · i∈0 .. 490
      Buffer_wLoc ( new ) := 0
      Buffer_rLoc ( new ) := 0
      Buffer := Buffer ∪ {new}
      CommBufferSimple_wBuff1(self) := new
      CommBufferSimple_state(self) := m4
  END

CommBufferSimple_m4 ≙
    ANY self, new
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      new ∈ Buffer_Set ∖ Buffer
      CommBufferSimple_state(self) = m4
    THEN
      Buffer_buff(new) := λ i · i∈0 .. 490
      Buffer_wLoc ( new ) := 0
      Buffer_rLoc ( new ) := 0
      Buffer := Buffer ∪ {new}
      CommBufferSimple_wBuff2(self) := new
      CommBufferSimple_state(self) := m5
  END

CommBufferSimple_m5 ≙
    REFINES newChan
    ANY self, new
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      new ∈ Channel_Set ∖ Channel
```

```
      CommBufferSimple_state(self) = m5
    THEN
      Channel_buff(new) :=   λ i · i∈0 .. 490
      Channel_wLoc ( new ) := 0
      Channel_rLoc ( new ) := 0
      Channel_rPID ( new ) := - 1
      Channel_wPID ( new ) := - 1
      Channel_writeSize ( new ) := - 1
      Channel := Channel ∪ {new}
      CommBufferSimple_chan(self) := new
      CommBufferSimple_state(self) := m6
  END

CommBufferSimple_while_m6 ≙
    ANY self, target
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      CommBufferSimple_state(self) = m6
      CommBufferSimple_v(self) < 16
      self ∈ dom( CommBufferSimple_wBuff1 )
      target = CommBufferSimple_wBuff1(self)
      Buffer_wLoc ( target ) < 49
      Buffer_wLoc ( target ) ≥ 0
    THEN
      Buffer_buff ( target ) :=
             Buffer_buff ( target ) ⩤
                    {Buffer_wLoc ( target ) ↦ CommBufferSimple_v ( self )}
      Buffer_wLoc ( target ) := Buffer_wLoc ( target ) + 1
      CommBufferSimple_state(self) := m7
  END

CommBufferSimple_while_m6_isNull ≙
    ANY self
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      CommBufferSimple_state(self) = m6
      CommBufferSimple_v(self) < 16
       ¬ ( self ∈ dom( CommBufferSimple_wBuff1 ) )
    THEN
```

```
      CommBufferSimple_state(self) := terminatedCommBufferSimple
  END

CommBufferSimple_m7 ≙
    ANY self, target
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      CommBufferSimple_state(self) = m7
      self ∈ dom( CommBufferSimple_wBuff2 )
      target = CommBufferSimple_wBuff2(self)
      Buffer_wLoc ( target ) < 49
      Buffer_wLoc ( target ) ≥ 0
    THEN
      Buffer_buff ( target ) :=
           Buffer_buff ( target ) ⩤
              {Buffer_wLoc ( target ) ↦CommBufferSimple_v (self) ∗ 2}
      Buffer_wLoc ( target ) := Buffer_wLoc ( target ) + 1
      CommBufferSimple_state(self) := m8
  END

CommBufferSimple_m7_isNull ≙
    ANY self
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      CommBufferSimple_state(self) = m7
       ¬ ( self ∈ dom( CommBufferSimple_wBuff2 ) )
    THEN
      CommBufferSimple_state(self) := terminatedCommBufferSimple
  END

CommBufferSimple_m8 ≙
    ANY self
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      CommBufferSimple_state(self) = m8
    THEN
      CommBufferSimple_v ( self ) := CommBufferSimple_v ( self ) + 1
      CommBufferSimple_state(self) := m6
```

```
  END

CommBufferSimple_while_m6_false  ≙
    ANY self
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      CommBufferSimple_state(self) = m6
       ¬ ( CommBufferSimple_v(self) < 16  )
    THEN
      CommBufferSimple_state(self)  :=  m9
  END

CommBufferSimple_m9  ≙
    REFINES newProc
    ANY self, new
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      new ∈ Proc_Set  \  Proc
      CommBufferSimple_state(self) = m9
      self ∈ dom(CommBufferSimple_wBuff1)
      self ∈ dom(CommBufferSimple_chan)
      1  ∉  ran(Proc_id)
    WITH
      p = new
      b = Buffer_buff(CommBufferSimple_wBuff1 ( self ))
    THEN
      Proc_id ( new )  :=  1
      Proc_buff ( new )  :=  CommBufferSimple_wBuff1 ( self )
      Proc_isWriter ( new )  :=  TRUE
      Proc_c ( new )  :=  CommBufferSimple_chan ( self )
      Proc_tmpBuffSz ( new )  :=  - 1
      Proc_tmpDat ( new )  :=  - 1
      Proc_state(new) := p1
      Proc  :=  Proc  ∪  {new}
      CommBufferSimple_wProc1(self)  :=  new
      CommBufferSimple_state(self)  :=  m10
  END

CommBufferSimple_m10  ≙
```

```
    REFINES newProc
    ANY self, new
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      new ∈ Proc_Set  \  Proc
      CommBufferSimple_state(self) = m10
      self ∈ dom(CommBufferSimple_wBuff2)
      self ∈ dom(CommBufferSimple_chan)
      2  ∉  ran(Proc_id)
    WITH
      p = new
      b = Buffer_buff(CommBufferSimple_wBuff2 ( self ))
    THEN
      Proc_id ( new )  :=  2
      Proc_buff ( new )  :=  CommBufferSimple_wBuff2 ( self )
      Proc_isWriter ( new )  :=  TRUE
      Proc_c ( new )  :=  CommBufferSimple_chan ( self )
      Proc_tmpBuffSz ( new )  :=  - 1
      Proc_tmpDat ( new )  :=  - 1
      Proc_state(new) := p1
      Proc  :=  Proc  ∪  {new}
      CommBufferSimple_wProc2(self)  :=  new
      CommBufferSimple_state(self)  :=  m11
  END

CommBufferSimple_m11  ≙
    REFINES newProc
    ANY self, new
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      new ∈ Proc_Set  \  Proc
      CommBufferSimple_state(self) = m11
      self ∈ dom(CommBufferSimple_rBuff1)
      self ∈ dom(CommBufferSimple_chan)
      3  ∉  ran(Proc_id)
    WITH
      p = new
      b = Buffer_buff(CommBufferSimple_rBuff1 ( self ))
    THEN
```

```
        Proc_id ( new ) := 3
        Proc_buff ( new ) := CommBufferSimple_rBuff1 ( self )
        Proc_isWriter ( new ) := FALSE
        Proc_c ( new ) := CommBufferSimple_chan ( self )
        Proc_tmpBuffSz ( new ) := - 1
        Proc_tmpDat ( new ) := - 1
        Proc_state(new) := p1
        Proc := Proc ∪ {new}
        CommBufferSimple_rProc1(self) := new
        CommBufferSimple_state(self) := m12
    END

  CommBufferSimple_m12 ≙
    REFINES newProc
    ANY self, new
    WHERE
      self ∈ CommBufferSimple
      self ∈ dom(CommBufferSimple_state)
      new ∈ Proc_Set \ Proc
      CommBufferSimple_state(self) = m12
      self ∈ dom(CommBufferSimple_rBuff2)
      self ∈ dom(CommBufferSimple_chan)
      4 ∉ ran(Proc_id)
    WITH
      p = new
      b = Buffer_buff(CommBufferSimple_rBuff2 ( self ))
    THEN
      Proc_id ( new ) := 4
      Proc_buff ( new ) := CommBufferSimple_rBuff2 ( self )
      Proc_isWriter ( new ) := FALSE
      Proc_c ( new ) := CommBufferSimple_chan ( self )
      Proc_tmpBuffSz ( new ) := - 1
      Proc_tmpDat ( new ) := - 1
      Proc_state(new) := p1
      Proc := Proc ∪ {new}
      CommBufferSimple_rProc2(self) := new
      CommBufferSimple_state(self) := terminatedCommBufferSimple
    END
END
```

Appendix D

Case Study 2 - OCB and Event-B Models, and Code

D.1 MainClass Specification

```
MainClass FFS2{
 // Attributes.
 DOStore doStore, OpenFileStore openFileStore,
 UserAppCreateFile userAppCreateFile, UserAppWriteFile userAppWriteFile,
 UserAppReadFile userAppReadFile, ErrorLog errorLog

 // Program entry point.
Operation main(){
   s1: doStore:=DOStore.create();
   s2: openFileStore:=OpenFileStore.create();
   s3: errorLog:=ErrorLog.create();
   s4: userAppCreateFile:=UserAppCreateFile.create(doStore,openFileStore,
     errorLog);
   s5: userAppWriteFile:=UserAppWriteFile.create(openFileStore,errorLog);
   s6: userAppReadFile:=UserAppReadFile.create(openFileStore,errorLog)
 }
}
```

D.2 ProcessClass CreateFile Specification

```
ProcessClass CreateFile{
 // Attributes.
```

```
 DataObject newFile, DOStore doStore, Integer doSpace,
 Integer openflSpace, OpenFileInfo openFileInfo,
 OpenFileStore openFileStore, Integer id, FileDirInfo fileDirInfo,
 Integer oMode, Integer sMode, Integer aMode, DataObject tmpObj,
 Integer tmpName, Boolean idFound, Integer index, ErrorLog errorLog

 // Constructor procedure.
Procedure create(DOStore doStor,OpenFileStore openFileStor,
                 Integer nameID, Integer oMde,Integer sMde,
                 ErrorLog errorLg){
 doSpace:=-1||openflSpace:=-1||doStore:=doStor||
 openFileStore:=openFileStor|| id:=nameID||oMode:=oMde||sMode:=sMde||
 aMode:=-1||tmpName:=-1||idFound:=FALSE||index:=0||errorLog:=errorLg
}

 // Description of the process' behaviour.
Operation run(){
 cf1: doSpace:=doStore.getSize();
 cf2: while(index<doSpace & idFound=FALSE) do
   tmpObj:=doStore.getAtIndex(index) andthen
   cf3: fileDirInfo:=tmpObj.getFileDirInfo();
   cf4: tmpName:=fileDirInfo.getID();
   cf5: if(tmpName=id) then idFound:=TRUE endif ;
   cf6: index:=index+1 endwhile ;
 cf7: if(oMode=1 & idFound=FALSE) then aMode:=2 andthen
   cf8: openflSpace:=openFileStore.reserveSpace();
   cf9: if(openflSpace>0) then doSpace:=doStore.reserveSpace() andthen
     cf10: if(doSpace>0) then fileDirInfo:=FileDirInfo.create(id) andthen
       // Clause cf11 refines create
       cf11: newFile:=DataObject.create(128,fileDirInfo);
       // Clause cf12 refines open_rw
       cf12: openFileInfo:=OpenFileInfo.create(aMode,sMode,newFile);
       cf13: doStore.add(newFile);
       cf14: openFileStore.add(openFileInfo) endif
     else openFileStore.unReserve() andthen
       cf15: doStore.unReserve();
       cf16: errorLog.add(6) endelse  endif
   else openFileStore.unReserve() andthen
     cf17: errorLog.add(5) endelse  endif
   elseif(oMode/=1) then errorLog.add(3) endelseif
   elseif(idFound=TRUE) then errorLog.add(2) endelseif
```

```
 }
}
```

D.3 ProcessClass WriteFile Specification

```
ProcessClass WriteFile{
 // Attributes.
 OpenFileStore openFileStore, UserBuffer buffer, Integer id,
 Integer tmpName, OpenFileInfo file, Integer bytes, Integer index,
 Integer openFileCnt, Boolean fileFound, FileDirInfo fileDirInfo,
 Integer data, DataObject dataObject, Integer offset, Integer aMode,
 ErrorLog errorLog, Integer freeSpace

 // Constructor procedure.
Procedure create(OpenFileStore openFileStor,Integer fName,
                 UserBuffer buffr,Integer byts,ErrorLog errorLg){
 openFileStore:=openFileStor||id:=fName||buffer:=buffr||bytes:=byts||
 index:=0||openFileCnt:=0||fileFound:=FALSE||tmpName:=-1||data:=-1||
 offset:=0||aMode:=-1||errorLog:=errorLg||freeSpace:=0
}

 // Description of the process' behaviour.
Operation run(){
 wf1: openFileCnt:=openFileStore.getSize();
 wf2: while(index<openFileCnt & fileFound=FALSE) do
  file:=openFileStore.getAtIndex(index) andthen
  wf3: dataObject:=file.getDataObject();
  wf4: fileDirInfo:=dataObject.getFileDirInfo();
  wf5: tmpName:=fileDirInfo.getID();
  wf6: if(tmpName=id) then fileFound:=TRUE endif ;
  wf7: index:=index+1 endwhile ;
  wf8: if(fileFound=TRUE) then aMode:=file.getAccessMode() andthen
    wf9: if(aMode=1 or aMode=2) then
      freeSpace:=dataObject.reserveSpace() andthen
      // Clause wf10 refines w_start
      wf10: if(freeSpace>0) then index:=0 andthen
        wf11: file.resetOffset();
        // Clause wf12_false, where ¬(index < bytes), refines w_end
        wf12: while(index<bytes) do data:=buffer.get(index) andthen
          wf13: offset:=file.getOffset();
```

```
        // Clause wf14 refines w_step
        wf14: dataObject.write(data, offset);
        wf15: index:=index+1;
        wf16: file.incOffset() endwhile  endif
     else dataObject.unReserve() andthen
      wf17: errorLog.add(7) endelse  endif
     else errorLog.add(4) endelse  endif
  else errorLog.add(1) endelse
 }
}
```

D.4 ProcessClass ReadFile Specification

```
ProcessClass ReadFile{
 // Attributes.
 OpenFileStore openFileStore, UserBuffer buffer, Integer id,
 Integer tmpName, OpenFileInfo file, Integer bytes, Integer index,
 Integer openFileCnt, Boolean fileFound, FileDirInfo fileDirInfo,
 Integer data, DataObject dataObject, Integer offset, Integer aMode,
 ErrorLog errorLog

 // Constructor procedure.
 Procedure create(OpenFileStore openFileStor, Integer fName,
                  UserBuffer buffr, Integer byts, ErrorLog errorLg){
  openFileStore:=openFileStor||id:=fName||buffer:=buffr||
  bytes:=byts||index:=0||openFileCnt:=0||fileFound:=FALSE||
  tmpName:=-1||data:=-1||offset:=0||aMode:=-1||errorLog:=errorLg
 }

 // Description of the process' behaviour.
 Operation run(){
  rf1: openFileCnt:=openFileStore.getSize();
  rf2: while(index<openFileCnt & fileFound=FALSE) do
    file:=openFileStore.getAtIndex(index) andthen
    rf3: dataObject:=file.getDataObject();
    rf4: fileDirInfo:=dataObject.getFileDirInfo();
    rf5: tmpName:=fileDirInfo.getID();
    rf6: if(tmpName=id) then fileFound:=TRUE endif ;
    rf7: index:=index+1 endwhile ;
  rf8: if(fileFound=TRUE) then aMode:=file.getAccessMode() andthen
```

```
    // Clause rf9 refines r_start
    rf9: if(aMode=0 or aMode=2) then index:=0 andthen
      rf10: file.resetOffset();
      // Clause rf11_false, where ¬(index < bytes), refines r_end
      rf11: while(index<bytes) do offset:=file.getOffset() andthen
        rf12: data:=dataObject.read(offset);
        // Clause rf13 refines r_step
        rf13: buffer.add(data);
        rf14: index:=index+1;
        rf15: file.incOffset() endwhile  endif
    else errorLog.add(4) endelse  endif
  else errorLog.add(1) endelse
 }
}
```

D.5 ProcessClass UserAppCreateFile Specification

```
ProcessClass UserAppCreateFile{
 // Attributes.
 CreateFile createFile, DOStore doStore,
 OpenFileStore openFileStore, ErrorLog errorLog

 // Constructor procedure.
 Procedure create(DOStore doStor,OpenFileStore openFileStor,
            ErrorLog errorLg){
  doStore:=doStor||openFileStore:=openFileStor||errorLog:=errorLg
 }

 // Description of the process' behaviour.
 Operation run(){
  ua1: createFile:=
            CreateFile.create(doStore,openFileStore,0,1,0,errorLog)
 }
}
```

D.6 ProcessClass UserAppWriteFile Specification

```
ProcessClass UserAppWriteFile{
 // Attributes.
```

```
 OpenFileStore openFileStore, UserBuffer buff1, WriteFile writeFile1,
 Integer data, ErrorLog errorLog

 // Constructor procedure.
 Procedure create(OpenFileStore openFileStor,ErrorLog errorLg){
  openFileStore:=openFileStor|| data:=65||errorLog:=errorLg
 }

 // Description of process' behaviour.
 Operation run(){
  // Clause uaw1 refines makeUWBuff
  uaw1: buff1:=UserBuffer.create();
  uaw2: while(data<70) do buff1.add(data) andthen
    uaw3: data:=data+1 endwhile ;
  uaw4: writeFile1:=WriteFile.create(openFileStore,0,buff1,5,errorLog)
 }
}
```

D.7 ProcessClass UserAppReadFile Specification

```
ProcessClass UserAppReadFile{
 // Attributes.
 OpenFileStore openFileStore, UserBuffer buff1, ReadFile readFile1,
 ErrorLog errorLog

 // Constructor procedure.
 Procedure create(OpenFileStore openFileStor,ErrorLog errorLg){
  openFileStore:=openFileStor||errorLog:=errorLg
 }

 // Description of process' behaviour.
 Operation run(){
  uar1: buff1:=UserBuffer.create();
  uar2: readFile1:=ReadFile.create(openFileStore,0,buff1,5,errorLog)
 }
}
```

D.8 MonitorClass FileDirInfo Specification

```
MonitorClass FileDirInfo{
 // Attributes.
 Integer fileOffset, Integer id

 // Constructor procedure.
 Procedure create(Integer nameID){
  fileOffset:=0 || id:=nameID
 }

 // Return file ID associated with this FileDirInfo.
 Procedure getID(){
  return:=id
 }: Integer
}
```

D.9 MonitorClass DataObject Specification

```
MonitorClass DataObject{
 // Attributes.
 Integer type, FileDirInfo fileDirInfo, Integer[10] data,
 Integer freeSpace

 // Constructor procedure.
 Procedure create(Integer typ,FileDirInfo fileDirInf){
  type:=typ || fileDirInfo:=fileDirInf||freeSpace:=10
 }

 // Return FileDirInfo associated with this data object.
 Procedure getFileDirInfo(){
  when(fileDirInfo /= null){
   return:=fileDirInfo }
 }: FileDirInfo

 // Return the byte at the specified offset.
 Procedure read(Integer offset){
  when(offset>=0 & offset < 10){
   return:=data[offset] }
 }: Integer
```

```
 // Procedure called in clause wf14 - refines w_step
 // Write the value 'val' at the specified offset.
 Procedure write(Integer val, Integer offset){
  when(offset>=0 & offset < 10){
    data[offset]:=val }
 }


 // Return this data object's type.
 Procedure getType(){
  return:=type
 }: Integer

 // Reserve space to write a byte in this data object.
 Procedure reserveSpace(){
  return :=freeSpace||freeSpace:=freeSpace-1
 }: Integer

 // Free a previously reserved space in this data object.
 Procedure unReserve(){
  freeSpace:=freeSpace+1
 }
}
```

D.10 MonitorClass OpenFileInfo Specification

```
MonitorClass OpenFileInfo{
 // Attributes.
 Integer accessMode, Integer shareMode, Integer fileOffset,
 DataObject dataObject

 // Constructor procedure.
 Procedure create(Integer aMode, Integer sMode,
       DataObject dataObj){
  shareMode:=sMode || accessMode:=aMode || fileOffset:=0 ||
  dataObject:=dataObj
 }

 // Return the current offset of the file associated with OpenFileInfo.
```

```
 Procedure getOffset(){
   return:=fileOffset
 }: Integer

 // Return the data object associated with OpenFileInfo.
 Procedure getDataObject(){
   when(dataObject /=null){
     return:=dataObject }
 }: DataObject

 // Reset the offset of the file associated with OpenFileInfo.
 Procedure resetOffset(){
   fileOffset:=0
}

 // Increment the current offset of the file associated with OpenFileInfo.
 Procedure incOffset(){
   fileOffset:=fileOffset+1
 }

 // Return the access mode of the file associated with OpenFileInfo.
 Procedure getAccessMode(){
    return:=accessMode
 }: Integer

 // Return the share mode of the file associated with OpenFileInfo.
 Procedure getShareMode(){
    return:=shareMode
 }: Integer
}
```

D.11 MonitorClass DOStore Specification

```
MonitorClass DOStore{
 // Attributes.
 DataObject[5] doArray, Integer size, Integer capacity,
 Integer freeSpace

 // Constructor procedure.
 Procedure create(){
```

```
  size:=0 || capacity:=5 || freeSpace:=5
 }

 Procedure add(DataObject f){
  when(size>=0 & size<capacity & capacity=5){
   doArray[size]:=f ||  size:=size+1 }
 }

 Procedure getAtIndex(Integer indx){
  when(indx>=0 & indx<size & doArray[indx] /= null){
   return:=doArray[indx] }
 }: DataObject

 Procedure reserveSpace(){
  return:=freeSpace || freeSpace:=freeSpace-1
 }: Integer

 Procedure unReserve(){
  freeSpace:=freeSpace+1
 }

 Procedure getSize(){
  return:=size
 }: Integer
}
```

D.12 MonitorClass OpenFileStore Specification

```
MonitorClass OpenFileStore{
 // Attributes.
 OpenFileInfo[5] openArray, Integer size, Integer capacity,
 Integer freeSpace

 // Constructor procedure.
 Procedure create(){
  size:=0 || capacity:=5 || freeSpace:=5
 }
 // add the OpenFileInfo object to the OpenFileStore array.

 Procedure add(OpenFileInfo f){
```

```
  when(size>=0 & size<capacity & capacity = 5){
    openArray[size]:=f ||  size:=size+1 }
 }

 // Return the OpenFileInfo object at the given array index.
 Procedure getAtIndex(Integer indx){
  when(indx>=0 & indx<size & openArray[indx] /= null){
    return:=openArray[indx] }
 }: OpenFileInfo

 // Reserve space in the OpenFileStore for an OpenFileInfo Object.
 Procedure reserveSpace(){
  return:=freeSpace || freeSpace:=freeSpace-1
 }: Integer

 // Free up previously reserved space in the OpenFileStore.
 Procedure unReserve(){
  freeSpace:=freeSpace+1
 }

 // Return the number of OpenFileInfo objects in the OpenFileStore.
 Procedure getSize(){
  return:=size
 }: Integer
}
```

D.13 MonitorClass UserBuffer Specification

```
MonitorClass UserBuffer{
 // Attributes.
 Integer[10] buffer, Integer capacity, Integer size

 // Constructor Procedure.
 Procedure create(){
  capacity:=10 || size:=0
 }

 // Procedure called in clause rf13 - refines r_step
 // Add a value 'val' to the UserBuffer array.
 Procedure add(Integer val){
```

```
  when(size>=0 & size<capacity & capacity=10){
   buffer[size]:=val ||size:=size+1 }
 }

 // Return the value at the given index.
 Procedure get(Integer indx){
  when(indx>=0 & indx<capacity & capacity=10){
   return:=buffer[indx] }
 }: Integer
}
```

D.14 MonitorClass ErrorLog Specification

```
 MonitorClass ErrorLog{
  // Attributes.
 Integer[5] error, Integer size, Integer lastIndex

 // Constructor Procedure.
 Procedure create(){
  size:=0 || lastIndex:=-1
 }

 // Add an error code to the ErrorLog.
 Procedure add(Integer errorCode){
  when(size>=0 & size<5){
   error[size]:=errorCode || size:=size+1 || lastIndex:=size }
 }

 // Get the last error recorded.
 Procedure getLast(){
    return:=error[lastIndex]
 }: Integer

 // Remove the last error from the log.
 Procedure removeLast(){
  when(size>0 & size<=5){
   error[lastIndex]:=0||size:=size-1||lastIndex:=lastIndex-1 }
 }
}
```

D.15 The Flash File System Event-B Implementation Model

```
MACHINE
FFS2

SEES
FFS2_CTX

VARIABLES
FileDirInfo, FileDirInfo_fileOffset, FileDirInfo_id, DataObject,
DataObject_type, DataObject_fileDirInfo, DataObject_data,
DataObject_freeSpace, OpenFileInfo, OpenFileInfo_accessMode,
OpenFileInfo_shareMode, OpenFileInfo_fileOffset,
OpenFileInfo_dataObject, DOStore, DOStore_doArray, DOStore_size,
DOStore_capacity, DOStore_freeSpace, OpenFileStore,
OpenFileStore_openArray, OpenFileStore_size,
OpenFileStore_capacity, OpenFileStore_freeSpace, UserBuffer,
UserBuffer_buffer, UserBuffer_capacity, UserBuffer_size,
ErrorLog, ErrorLog_error, ErrorLog_size, ErrorLog_lastIndex,
CreateFile, CreateFile_state, CreateFile_newFile, CreateFile_doStore,
CreateFile_doSpace, CreateFile_openflSpace, CreateFile_openFileInfo,
CreateFile_openFileStore, CreateFile_id, CreateFile_fileDirInfo,
CreateFile_oMode, CreateFile_sMode, CreateFile_aMode, CreateFile_tmpObj,
CreateFile_tmpName, CreateFile_idFound, CreateFile_index,
CreateFile_errorLog, WriteFile, WriteFile_state, WriteFile_openFileStore,
WriteFile_buffer, WriteFile_id, WriteFile_tmpName, WriteFile_file
WriteFile_bytes, WriteFile_index, WriteFile_openFileCnt,
WriteFile_fileFound, WriteFile_fileDirInfo, WriteFile_data,
WriteFile_dataObject, WriteFile_offset, WriteFile_aMode,
WriteFile_errorLog, WriteFile_freeSpace, ReadFile, ReadFile_state
ReadFile_openFileStore, ReadFile_buffer, ReadFile_id,
ReadFile_tmpName, ReadFile_file, ReadFile_bytes, ReadFile_index,
ReadFile_openFileCnt, ReadFile_fileFound, ReadFile_fileDirInfo,
ReadFile_data, ReadFile_dataObject, ReadFile_offset,
ReadFile_aMode, ReadFile_errorLog, UserAppCreateFile,
UserAppCreateFile_state, UserAppCreateFile_createFile,
UserAppCreateFile_doStore, UserAppCreateFile_openFileStore,
UserAppCreateFile_errorLog, UserAppWriteFile,
UserAppWriteFile_state, UserAppWriteFile_openFileStore,
UserAppWriteFile_buff1, UserAppWriteFile_writeFile1,
UserAppWriteFile_data, UserAppWriteFile_errorLog,
```

UserAppReadFile, UserAppReadFile_state, UserAppReadFile_openFileStore,
UserAppReadFile_buff1, UserAppReadFile_readFile1,
UserAppReadFile_errorLog, FFS2, FFS2_doStore, FFS2_openFileStore,
FFS2_userAppCreateFile, FFS2_userAppWriteFile, FFS2_userAppReadFile
FFS2_errorLog, FFS2_state

INVARIANTS
FileDirInfo: FileDirInfo $\in \mathbb{P}$(FileDirInfo_Set)
FileDirInfo_fileOffset: FileDirInfo_fileOffset $\in$ FileDirInfo $\rightarrow \mathbb{Z}$
FileDirInfo_id: FileDirInfo_id $\in$ FileDirInfo $\rightarrow \mathbb{Z}$
DataObject: DataObject $\in \mathbb{P}$(DataObject_Set)
DataObject_type: DataObject_type $\in$ DataObject $\rightarrow \mathbb{Z}$
DataObject_fileDirInfo: DataObject_fileDirInfo $\in$
DataObject $\nrightarrow$ FileDirInfo
DataObject_data: DataObject_data $\in$ DataObject $\rightarrow$ (0 .. 9 $\rightarrow \mathbb{Z}$)
DataObject_freeSpace: DataObject_freeSpace $\in$ DataObject $\rightarrow \mathbb{Z}$
OpenFileInfo: OpenFileInfo $\in \mathbb{P}$(OpenFileInfo_Set)
OpenFileInfo_accessMode: OpenFileInfo_accessMode $\in$ OpenFileInfo $\rightarrow \mathbb{Z}$
OpenFileInfo_shareMode: OpenFileInfo_shareMode $\in$ OpenFileInfo $\rightarrow \mathbb{Z}$
OpenFileInfo_fileOffset: OpenFileInfo_fileOffset $\in$ OpenFileInfo $\rightarrow \mathbb{Z}$
OpenFileInfo_dataObject: OpenFileInfo_dataObject $\in$
OpenFileInfo $\nrightarrow$ DataObject
DOStore: DOStore $\in \mathbb{P}$(DOStore_Set)
DOStore_doArray: DOStore_doArray $\in$ DOStore $\rightarrow$ (0 .. 4 $\nrightarrow$ DataObject)
DOStore_size: DOStore_size $\in$DOStore $\rightarrow \mathbb{Z}$
DOStore_capacity: DOStore_capacity $\in$ DOStore $\rightarrow \mathbb{Z}$
DOStore_freeSpace: DOStore_freeSpace $\in$ DOStore DOStore $\rightarrow \mathbb{Z}$
OpenFileStore: OpenFileStore $\in \mathbb{P}$(OpenFileStore_Set)
OpenFileStore_openArray: OpenFileStore_openArray $\in$
DOStore OpenFileStore $\rightarrow$ (0 .. 4 $\nrightarrow$ OpenFileInfo)
OpenFileStore_size: OpenFileStore_size $\in$ OpenFileStore $\rightarrow \mathbb{Z}$
OpenFileStore_capacity: OpenFileStore_capacity $\in$ OpenFileStore $\rightarrow \mathbb{Z}$
OpenFileStore_freeSpace: OpenFileStore_freeSpace $\in$ OpenFileStore $\rightarrow \mathbb{Z}$
UserBuffer: UserBuffer $\in \mathbb{P}$(UserBuffer_Set)
UserBuffer_buffer: UserBuffer_buffer $\in$ UserBuffer $\rightarrow$ (0 .. 9 $\rightarrow \mathbb{Z}$)
UserBuffer_capacity: UserBuffer_capacity $\in$ UserBuffer $\rightarrow \mathbb{Z}$
UserBuffer_size: UserBuffer_size $\in$ UserBuffer $\rightarrow \mathbb{Z}$
ErrorLog: ErrorLog $\in \mathbb{P}$(ErrorLog_Set)
ErrorLog_error: ErrorLog_error $\in$ ErrorLog $\rightarrow$ (0 .. 4 $\rightarrow \mathbb{Z}$)
ErrorLog_size: ErrorLog_size $\in$ ErrorLog $\rightarrow \mathbb{Z}$
ErrorLog_lastIndex: ErrorLog_lastIndex $\in$ ErrorLog $\rightarrow \mathbb{Z}$

CreateFile: CreateFile ∈ ℙ(CreateFile_Set)
CreateFile_state: CreateFile_state ∈ CreateFile ⇸ CreateFile_states
CreateFile_newFile: CreateFile_newFile ∈ CreateFile ⇸ DataObject
CreateFile_doStore: CreateFile_doStore ∈ CreateFile ⇸ DOStore
CreateFile_doSpace: CreateFile_doSpace ∈ CreateFile → ℤ
CreateFile_openflSpace: CreateFile_openflSpace ∈ CreateFile → ℤ
CreateFile_openFileInfo: CreateFile_openFileInfo ∈
CreateFile ⇸ OpenFileInfo
CreateFile_openFileStore: CreateFile_openFileStore ∈
CreateFile ⇸ OpenFileStore
CreateFile_id: CreateFile_id ∈ CreateFile → ℤ
CreateFile_fileDirInfo: CreateFile_fileDirInfo ∈
CreateFile ⇸ FileDirInfo
CreateFile_oMode: CreateFile_oMode ∈ CreateFile → ℤ
CreateFile_sMode: CreateFile_sMode ∈ CreateFile → ℤ
CreateFile_aMode: CreateFile_aMode ∈ CreateFile → ℤ
CreateFile_tmpObj: CreateFile_tmpObj ∈ CreateFile ⇸ DataObject
CreateFile_tmpName: CreateFile_tmpName ∈ CreateFile → ℤ
CreateFile_idFound: CreateFile_idFound ∈ CreateFile → BOOL
CreateFile_index: CreateFile_index ∈ CreateFile → ℤ
CreateFile_errorLog: CreateFile_errorLog ∈ CreateFile ⇸ ErrorLog
WriteFile: WriteFile ∈ ℙ(WriteFile_Set)
WriteFile_state: WriteFile_state ∈ WriteFile ⇸ WriteFile_states
WriteFile_openFileStore: WriteFile_openFileStore ∈
WriteFile ⇸ OpenFileStore
WriteFile_buffer: WriteFile_buffer ∈ WriteFile ⇸ UserBuffer
WriteFile_id: WriteFile_id ∈ WriteFile → ℤ
WriteFile_tmpName: WriteFile_tmpName ∈ WriteFile → ℤ
WriteFile_file: WriteFile_file ∈ WriteFile ⇸ OpenFileInfo
WriteFile_bytes: WriteFile_bytes ∈ WriteFile → ℤ
WriteFile_index: WriteFile_index ∈ WriteFile → ℤ
WriteFile_openFileCnt: WriteFile_openFileCnt ∈ WriteFile → ℤ
WriteFile_fileFound: WriteFile_fileFound ∈ WriteFile → BOOL
WriteFile_fileDirInfo: WriteFile_fileDirInfo ∈ WriteFile ⇸ FileDirInfo
WriteFile_data: WriteFile_data ∈ WriteFile → ℤ
WriteFile_dataObject: WriteFile_dataObject ∈ WriteFile ⇸ DataObject
WriteFile_offset: WriteFile_offset ∈ WriteFile → ℤ
WriteFile_aMode: WriteFile_aMode ∈ WriteFile → ℤ
WriteFile_errorLog: WriteFile_errorLog ∈ WriteFile ⇸ ErrorLog
WriteFile_freeSpace: WriteFile_freeSpace ∈ WriteFile → ℤ
ReadFile: ReadFile ∈ ℙ(ReadFile_Set)

ReadFile_state: ReadFile_state ∈ ReadFile ⇸ ReadFile_states
ReadFile_openFileStore: ReadFile_openFileStore ∈
ReadFile ⇸ OpenFileStore
ReadFile_buffer: ReadFile_buffer ∈ ReadFile ⇸ UserBuffer
ReadFile_id: ReadFile_id ∈ ReadFile → ℤ
ReadFile_tmpName: ReadFile_tmpName ∈ ReadFile → ℤ
ReadFile_file: ReadFile_file ∈ ReadFile ⇸ OpenFileInfo
ReadFile_bytes: ReadFile_bytes ∈ ReadFile → ℤ
ReadFile_index: ReadFile_index ∈ ReadFile → ℤ
ReadFile_openFileCnt: ReadFile_openFileCnt ∈ ReadFile → ℤ
ReadFile_fileFound: ReadFile_fileFound ∈ ReadFile → BOOL
ReadFile_fileDirInfo: ReadFile_fileDirInfo ∈ ReadFile ⇸ FileDirInfo
ReadFile_data: ReadFile_data ∈ ReadFile → ℤ
ReadFile_dataObject: ReadFile_dataObject ∈ ReadFile ⇸ DataObject
ReadFile_offset: ReadFile_offset ∈ ReadFile → ℤ
ReadFile_aMode: ReadFile_aMode ∈ ReadFile → ℤ
ReadFile_errorLog: ReadFile_errorLog ∈ ReadFile ⇸ ErrorLog
UserAppCreateFile: UserAppCreateFile ∈ ℙ(UserAppCreateFile_Set)
UserAppCreateFile_state: UserAppCreateFile_state ∈
UserAppCreateFile ⇸ UserAppCreateFile_states
UserAppCreateFile_createFile: UserAppCreateFile_createFile ∈
UserAppCreateFile ⇸ CreateFile
UserAppCreateFile_doStore: UserAppCreateFile_doStore ∈
UserAppCreateFile ⇸ DOStore
UserAppCreateFile_openFileStore: UserAppCreateFile_openFileStore ∈
UserAppCreateFile ⇸ OpenFileStore
UserAppCreateFile_errorLog: UserAppCreateFile_errorLog ∈
UserAppCreateFile ⇸ ErrorLog
UserAppWriteFile: UserAppWriteFile ∈ ℙ(UserAppWriteFile_Set)
UserAppWriteFile_state: UserAppWriteFile_state ∈
UserAppWriteFile ⇸ UserAppWriteFile_states
UserAppWriteFile_openFileStore: UserAppWriteFile_openFileStore ∈
UserAppWriteFile ⇸ OpenFileStore
UserAppWriteFile_buff1: UserAppWriteFile_buff1 ∈
UserAppWriteFile ⇸ UserBuffer
UserAppWriteFile_writeFile1: UserAppWriteFile_writeFile1 ∈
UserAppWriteFile ⇸ WriteFile
UserAppWriteFile_data: UserAppWriteFile_data ∈
UserAppWriteFile → ℤ
UserAppWriteFile_errorLog: UserAppWriteFile_errorLog ∈
UserAppWriteFile ⇸ ErrorLog

```
UserAppReadFile: UserAppReadFile ∈ ℙ(UserAppReadFile_Set)
UserAppReadFile_state: UserAppReadFile_state  ∈
        UserAppReadFile ⇸ UserAppReadFile_states
UserAppReadFile_openFileStore: UserAppReadFile_openFileStore ∈
        UserAppReadFile ⇸ OpenFileStore
UserAppReadFile_buff1: UserAppReadFile_buff1 ∈
        UserAppReadFile ⇸ UserBuffer
UserAppReadFile_readFile1: UserAppReadFile_readFile1 ∈
        UserAppReadFile ⇸ ReadFile
UserAppReadFile_errorLog: UserAppReadFile_errorLog ∈
        UserAppReadFile ⇸ ErrorLog
fFS2: fFS2  ∈ FFS2_Set
FFS2: FFS2  ∈ P(FFS2_Set)
FFS2_doStore: FFS2_doStore ∈ FFS2 ⇸ DOStore
FFS2_openFileStore: FFS2_openFileStore ∈ FFS2 ⇸ OpenFileStore
FFS2_userAppCreateFile: FFS2_userAppCreateFile ∈
        FFS2 ⇸ UserAppCreateFile
FFS2_userAppWriteFile: FFS2_userAppWriteFile ∈
        FFS2 ⇸ UserAppWriteFile
FFS2_userAppReadFile: FFS2_userAppReadFile ∈
        FFS2 ⇸ UserAppReadFile
FFS2_errorLog: FFS2_errorLog ∈ FFS2 ⇸ ErrorLog
FFS2_state: FFS2_state  ∈  FFS2 ⇸ FFS2_states

EVENTS

INITIALISATION  ≙

WHICH IS
ordinary

BEGIN
FileDirInfo: FileDirInfo := ∅
FileDirInfo_fileOffset: FileDirInfo_fileOffset := ∅
FileDirInfo_id: FileDirInfo_id := ∅
DataObject: DataObject := ∅
DataObject_type: DataObject_type := ∅
DataObject_fileDirInfo: DataObject_fileDirInfo := ∅
DataObject_data: DataObject_data := ∅
DataObject_freeSpace: DataObject_freeSpace := ∅
OpenFileInfo: OpenFileInfo := ∅
```

OpenFileInfo_accessMode: OpenFileInfo_accessMode := ∅
OpenFileInfo_shareMode: OpenFileInfo_shareMode := ∅
OpenFileInfo_fileOffset: OpenFileInfo_fileOffset := ∅
OpenFileInfo_dataObject: OpenFileInfo_dataObject := ∅
DOStore: DOStore := ∅
DOStore_doArray: DOStore_doArray := ∅
DOStore_size: DOStore_size := ∅
DOStore_capacity: DOStore_capacity := ∅
DOStore_freeSpace: DOStore_freeSpace := ∅
OpenFileStore: OpenFileStore := ∅
OpenFileStore_openArray: OpenFileStore_openArray := ∅
OpenFileStore_size: OpenFileStore_size := ∅
OpenFileStore_capacity: OpenFileStore_capacity := ∅
OpenFileStore_freeSpace: OpenFileStore_freeSpace := ∅
UserBuffer: UserBuffer := ∅
UserBuffer_buffer: UserBuffer_buffer := ∅
UserBuffer_capacity: UserBuffer_capacity := ∅
UserBuffer_size: UserBuffer_size := ∅
ErrorLog: ErrorLog := ∅
ErrorLog_error: ErrorLog_error := ∅
ErrorLog_size: ErrorLog_size := ∅
ErrorLog_lastIndex: ErrorLog_lastIndex := ∅
CreateFile: CreateFile := ∅
CreateFile_state: CreateFile_state := ∅
CreateFile_newFile: CreateFile_newFile := ∅
CreateFile_doStore: CreateFile_doStore := ∅
CreateFile_doSpace: CreateFile_doSpace := ∅
CreateFile_openflSpace: CreateFile_openflSpace := ∅
CreateFile_openFileInfo: CreateFile_openFileInfo := ∅
CreateFile_openFileStore: CreateFile_openFileStore := ∅
CreateFile_id: CreateFile_id := ∅
CreateFile_fileDirInfo: CreateFile_fileDirInfo := ∅
CreateFile_oMode: CreateFile_oMode := ∅
CreateFile_sMode: CreateFile_sMode := ∅
CreateFile_aMode: CreateFile_aMode := ∅
CreateFile_tmpObj: CreateFile_tmpObj := ∅
CreateFile_tmpName: CreateFile_tmpName := ∅
CreateFile_idFound: CreateFile_idFound := ∅
CreateFile_index: CreateFile_index := ∅
CreateFile_errorLog: CreateFile_errorLog := ∅
WriteFile: WriteFile := ∅

WriteFile_state: WriteFile_state := ∅
WriteFile_openFileStore: WriteFile_openFileStore := ∅
WriteFile_buffer: WriteFile_buffer := ∅
WriteFile_id: WriteFile_id := ∅
WriteFile_tmpName: WriteFile_tmpName := ∅
WriteFile_file: WriteFile_file := ∅
WriteFile_bytes: WriteFile_bytes := ∅
WriteFile_index: WriteFile_index := ∅
WriteFile_openFileCnt: WriteFile_openFileCnt := ∅
WriteFile_fileFound: WriteFile_fileFound := ∅
WriteFile_fileDirInfo: WriteFile_fileDirInfo := ∅
WriteFile_data: WriteFile_data := ∅
WriteFile_dataObject: WriteFile_dataObject := ∅
WriteFile_offset: WriteFile_offset := ∅
WriteFile_aMode: WriteFile_aMode := ∅
WriteFile_errorLog: WriteFile_errorLog := ∅
WriteFile_freeSpace: WriteFile_freeSpace := ∅
ReadFile: ReadFile := ∅
ReadFile_state: ReadFile_state := ∅
ReadFile_openFileStore: ReadFile_openFileStore := ∅
ReadFile_buffer: ReadFile_buffer := ∅
ReadFile_id: ReadFile_id := ∅
ReadFile_tmpName: ReadFile_tmpName := ∅
ReadFile_file: ReadFile_file := ∅
ReadFile_bytes: ReadFile_bytes := ∅
ReadFile_index: ReadFile_index := ∅
ReadFile_openFileCnt: ReadFile_openFileCnt := ∅
ReadFile_fileFound: ReadFile_fileFound := ∅
ReadFile_fileDirInfo: ReadFile_fileDirInfo := ∅
ReadFile_data: ReadFile_data := ∅
ReadFile_dataObject: ReadFile_dataObject := ∅
ReadFile_offset: ReadFile_offset := ∅
ReadFile_aMode: ReadFile_aMode := ∅
ReadFile_errorLog: ReadFile_errorLog := ∅
UserAppCreateFile: UserAppCreateFile := ∅
UserAppCreateFile_state: UserAppCreateFile_state := ∅
UserAppCreateFile_createFile: UserAppCreateFile_createFile := ∅
UserAppCreateFile_doStore: UserAppCreateFile_doStore := ∅
UserAppCreateFile_openFileStore: UserAppCreateFile_openFileStore := ∅
UserAppCreateFile_errorLog: UserAppCreateFile_errorLog := ∅
UserAppWriteFile: UserAppWriteFile := ∅

```
UserAppWriteFile_state: UserAppWriteFile_state  := ∅
UserAppWriteFile_openFileStore: UserAppWriteFile_openFileStore := ∅
UserAppWriteFile_buff1: UserAppWriteFile_buff1 := ∅
UserAppWriteFile_writeFile1: UserAppWriteFile_writeFile1 := ∅
UserAppWriteFile_data: UserAppWriteFile_data := ∅
UserAppWriteFile_errorLog: UserAppWriteFile_errorLog := ∅
UserAppReadFile: UserAppReadFile := ∅
UserAppReadFile_state: UserAppReadFile_state  := ∅
UserAppReadFile_openFileStore: UserAppReadFile_openFileStore := ∅
UserAppReadFile_buff1: UserAppReadFile_buff1 := ∅
UserAppReadFile_readFile1: UserAppReadFile_readFile1 := ∅
UserAppReadFile_errorLog: UserAppReadFile_errorLog := ∅
FFS2: FFS2 := ∅
FFS2_doStore: FFS2_doStore := ∅
FFS2_openFileStore: FFS2_openFileStore := ∅
FFS2_userAppCreateFile: FFS2_userAppCreateFile := ∅
FFS2_userAppWriteFile: FFS2_userAppWriteFile := ∅
FFS2_userAppReadFile: FFS2_userAppReadFile := ∅
FFS2_errorLog: FFS2_errorLog := ∅
FFS2_state: FFS2_state  := ∅

END

CreateFile_cf1   ≙

WHICH IS
ordinary
ANY
self
target
WHERE
label1: self ∈ CreateFile
label2: self ∈ dom(CreateFile_state)
label3: CreateFile_state(self) = cf1
label4: self ∈ dom( CreateFile_doStore )
label10: target = CreateFile_doStore(self)
THEN
label11: CreateFile_doSpace ( self )  := DOStore_size ( target )
label12: CreateFile_state(self) := cf2
END
```

```
CreateFile_cf1_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label5: self ∈ CreateFile
label6: self ∈ dom(CreateFile_state)
label7: CreateFile_state(self) = cf1
label8:  ¬( self ∈ dom( CreateFile_doStore ) )
THEN
label9: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_while_cf2  ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label13: self ∈ CreateFile
label14: self ∈ dom(CreateFile_state)
label15: CreateFile_state(self) = cf2
label16: CreateFile_index(self) < CreateFile_doSpace(self) ∧
          CreateFile_idFound(self) = FALSE
label17: self ∈ dom( CreateFile_doStore )
label24: target = CreateFile_doStore(self)
label26: target ∈dom( DOStore_doArray )
label27: CreateFile_index ( self ) ≥ 0
label28: CreateFile_index ( self ) < DOStore_size ( target )
label29: CreateFile_index ( self ) ∈ dom( DOStore_doArray ( target ) )
THEN
label25: CreateFile_tmpObj ( self ) :=
          DOStore_doArray ( target ) ( CreateFile_index ( self ) )
label30: CreateFile_state(self) := cf3
END

CreateFile_while_cf2_isNull  ≙
WHICH IS
ordinary
```

```
ANY
self
WHERE
label18: self ∈ CreateFile
label19: self ∈ dom(CreateFile_state)
label20: CreateFile_state(self) = cf2
label21: CreateFile_index(self) < CreateFile_doSpace(self) ∧
          CreateFile_idFound(self) = FALSE
label22:  ¬( self ∈ dom( CreateFile_doStore ) )
THEN
label23: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf3   ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label31: self ∈ CreateFile
label32: self ∈ dom(CreateFile_state)
label33: CreateFile_state(self) = cf3
label34: self ∈ dom( CreateFile_tmpObj )
label40: target = CreateFile_tmpObj(self)
label42: target ∈ dom( DataObject_fileDirInfo )
THEN
label41: CreateFile_fileDirInfo ( self ) :=
          DataObject_fileDirInfo ( target )
label43: CreateFile_state(self) := cf4
END

CreateFile_cf3_isNull   ≙
WHICH IS
ordinary
ANY
self
WHERE
label35: self ∈ CreateFile
label36: self ∈ dom(CreateFile_state)
label37: CreateFile_state(self) = cf3
```

```
label38:  ¬( self ∈ dom( CreateFile_tmpObj ) )
THEN
label39: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf4   ≙
WHICH IS
ordinary

ANY
self
target
WHERE
label44: self ∈ CreateFile
label45: self ∈ dom(CreateFile_state)
label46: CreateFile_state(self) = cf4
label47: self ∈ dom( CreateFile_fileDirInfo )
label53: target = CreateFile_fileDirInfo(self)
THEN
label54: CreateFile_tmpName ( self ) := FileDirInfo_id ( target )
label55: CreateFile_state(self) := cf5
END

CreateFile_cf4_isNull   ≙
WHICH IS
ordinary
ANY
self
WHERE
label48: self ∈ CreateFile
label49: self ∈ dom(CreateFile_state)
label50: CreateFile_state(self) = cf4
label51:  ¬( self ∈ dom( CreateFile_fileDirInfo ) )
THEN
label52: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf5   ≙
WHICH IS
ordinary
ANY
```

```
self
WHERE
label56: self ∈ CreateFile
label57: self ∈ dom(CreateFile_state)
label58: CreateFile_state(self) = cf5
label61: CreateFile_tmpName(self) = CreateFile_id(self)
THEN
label59: CreateFile_idFound ( self ) := TRUE
label60: CreateFile_state(self) := cf6
END

CreateFile_defaultElse_cf5  ≙
WHICH IS
ordinary
ANY
self
WHERE
label62: self ∈ CreateFile
label63: self ∈ dom(CreateFile_state)
label64: CreateFile_state(self) = cf5
label65:  ¬( CreateFile_tmpName(self) = CreateFile_id(self)  )
THEN
label66: CreateFile_state(self) := cf6
END

CreateFile_cf6  ≙
WHICH IS
ordinary
ANY
self
WHERE
label67: self ∈ CreateFile
label68: self ∈ dom(CreateFile_state)
label69: CreateFile_state(self) = cf6
THEN
label70: CreateFile_index ( self ) := CreateFile_index ( self ) + 1
label71: CreateFile_state(self) := cf2
END

CreateFile_while_cf2_false  ≙
WHICH IS
```

```
ordinary

ANY
self
WHERE
label72: self ∈ CreateFile
label73: self ∈ dom(CreateFile_state)
label74: CreateFile_state(self) = cf2
label75:  ¬( CreateFile_index(self) < CreateFile_doSpace(self) ∧
           CreateFile_idFound(self) = FALSE  )
THEN
label76: CreateFile_state(self) := cf7
END

CreateFile_cf7   ≙
WHICH IS
ordinary
ANY
self
WHERE
label77: self ∈ CreateFile
label78: self ∈ dom(CreateFile_state)
label79: CreateFile_state(self) = cf7
label82: CreateFile_oMode(self) = 1
label83: CreateFile_idFound(self) = FALSE
THEN
label80: CreateFile_aMode ( self ) := 2
label81: CreateFile_state(self) := cf8
END

CreateFile_cf8   ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label84: self ∈ CreateFile
label85: self ∈ dom(CreateFile_state)
label86: CreateFile_state(self) = cf8
label87: self ∈ dom( CreateFile_openFileStore )
```

```
label93: target = CreateFile_openFileStore(self)
THEN
label94: CreateFile_openflSpace ( self ) :=
          OpenFileStore_freeSpace ( target )
label95: OpenFileStore_freeSpace ( target ) :=
           OpenFileStore_freeSpace ( target ) - 1
label96: CreateFile_state(self) := cf9
END

CreateFile_cf8_isNull   ≙
WHICH IS
ordinary
ANY
self
WHERE
label88: self ∈ CreateFile
label89: self ∈ dom(CreateFile_state)
label90: CreateFile_state(self) = cf8
label91:  ¬( self ∈ dom( CreateFile_openFileStore ) )
THEN
label92: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf9   ≙

WHICH IS
ordinary
ANY
self
target
WHERE
label97: self ∈ CreateFile
label98: self ∈ dom(CreateFile_state)
label99: CreateFile_state(self) = cf9
label100: CreateFile_openflSpace(self) > 0
label101: self ∈ dom( CreateFile_doStore )
label108: target = CreateFile_doStore(self)
THEN
label109: CreateFile_doSpace ( self ) := DOStore_freeSpace ( target )
label110: DOStore_freeSpace ( target ) :=
            DOStore_freeSpace ( target ) - 1
```

label111: CreateFile_state(self) := cf10
END

CreateFile_cf9_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label102: self ∈ CreateFile
label103: self ∈ dom(CreateFile_state)
label104: CreateFile_state(self) = cf9
label105: CreateFile_openflSpace(self) > 0
label106: ¬(self ∈ dom(CreateFile_doStore))
THEN
label107: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf10 ≙
WHICH IS
ordinary
ANY
self
new
WHERE
label112: self ∈ CreateFile
label113: self ∈ dom(CreateFile_state)
label114: new ∈ FileDirInfo_Set \ FileDirInfo
label115: CreateFile_state(self) = cf10
label121: CreateFile_doSpace(self) > 0
THEN
label116: FileDirInfo_fileOffset (new) := 0
label117: FileDirInfo_id (new) := CreateFile_id (self)
label118: FileDirInfo := FileDirInfo ∪ {new}
label119: CreateFile_fileDirInfo(self) := new
label120: CreateFile_state(self) := cf11
END

CreateFile_cf11 ≙
WHICH IS
ordinary

```
ANY
self
new
WHERE
label122: self ∈ CreateFile
label123: self ∈ dom(CreateFile_state)
label124: new ∈ DataObject_Set \ DataObject
label125: CreateFile_state(self) = cf11
label127: self ∈ dom(CreateFile_fileDirInfo)
THEN
label126: DataObject_data(new):= λ i· i ∈ 0 .. 9|0
label128: DataObject_type ( new ) := 128
label129: DataObject_fileDirInfo ( new ) :=
            CreateFile_fileDirInfo ( self )
label130: DataObject_freeSpace ( new ) := 10
label131: DataObject := DataObject ∪ {new}
label132: CreateFile_newFile(self) := new
label133: CreateFile_state(self) := cf12
END

CreateFile_cf12   ≙
WHICH IS
ordinary
ANY
self
new
WHERE
label134: self ∈ CreateFile
label135: self ∈ dom(CreateFile_state)
label136: new ∈ OpenFileInfo_Set \ OpenFileInfo
label137: CreateFile_state(self) = cf12
label138: self ∈ dom(CreateFile_newFile)
THEN
label139: OpenFileInfo_shareMode ( new ) := CreateFile_sMode ( self )
label140: OpenFileInfo_accessMode ( new ) := CreateFile_aMode ( self )
label141: OpenFileInfo_fileOffset ( new ) := 0
label142: OpenFileInfo_dataObject ( new ) := CreateFile_newFile ( self )
label143: OpenFileInfo := OpenFileInfo ∪ {new}
label144: CreateFile_openFileInfo(self) := new
label145: CreateFile_state(self) := cf13
END
```

```
CreateFile_cf13   ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label146: self ∈ CreateFile
label147: self ∈ dom(CreateFile_state)
label148: CreateFile_state(self) = cf13
label149: self ∈ dom( CreateFile_doStore )
label155: target = CreateFile_doStore(self)
label156: self ∈ dom(CreateFile_newFile)
label159: DOStore_size ( target ) ≥ 0
label160: DOStore_size ( target ) < DOStore_capacity ( target )
label161: DOStore_capacity ( target ) = 5
THEN
label157: DOStore_doArray ( target ) :=
            DOStore_doArray ( target ) ⩤
              { DOStore_size ( target ) ↦ CreateFile_newFile ( self ) }
label158: DOStore_size ( target ) := DOStore_size ( target ) + 1
label162: CreateFile_state(self) := cf14
END

CreateFile_cf13_isNull   ≙
WHICH IS
ordinary
ANY
self
WHERE
label150: self ∈ CreateFile
label151: self ∈ dom(CreateFile_state)
label152: CreateFile_state(self) = cf13
label153:  ¬( self ∈ dom( CreateFile_doStore ) )
THEN
label154: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf14   ≙
WHICH IS
```

```
ordinary
ANY
self
target
WHERE
label163: self ∈ CreateFile
label164: self ∈ dom(CreateFile_state)
label165: CreateFile_state(self) = cf14
label166: self ∈ dom( CreateFile_openFileStore )
label172: target = CreateFile_openFileStore(self)
label173: self ∈ dom(CreateFile_openFileInfo)
label176: OpenFileStore_size ( target ) ≥ 0
label177: OpenFileStore_size ( target ) <
            OpenFileStore_capacity ( target )
label178: OpenFileStore_capacity ( target ) = 5
THEN
label174: OpenFileStore_openArray ( target ) :=
            OpenFileStore_openArray ( target ) ⩤
              { OpenFileStore_size ( target ) ↦ CreateFile_openFileInfo ( self ) }
label175: OpenFileStore_size ( target ) :=
           OpenFileStore_size ( target ) + 1
label179: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf14_isNull   ≙
WHICH IS
ordinary
ANY
self
WHERE
label167: self ∈ CreateFile
label168: self ∈ dom(CreateFile_state)
label169: CreateFile_state(self) = cf14
label170:  ¬( self ∈ dom( CreateFile_openFileStore ) )
THEN
label171: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf10_else   ≙
WHICH IS
ordinary
```

```
ANY
self
target
WHERE
label180: self ∈ CreateFile
label181: self ∈ dom(CreateFile_state)
label182: CreateFile_state(self) = cf10
label183:  ¬(CreateFile_doSpace(self) > 0 )
label184: self ∈ dom( CreateFile_openFileStore )
label191: target = CreateFile_openFileStore(self)
THEN
label192: OpenFileStore_freeSpace ( target ) :=
            OpenFileStore_freeSpace ( target ) + 1
label193: CreateFile_state(self) := cf15
END

CreateFile_cf10_else_isNull   ≙
WHICH IS
ordinary
ANY
self
WHERE
label185: self ∈ CreateFile
label186: self ∈ dom(CreateFile_state)
label187: CreateFile_state(self) = cf10
label188:  ¬(CreateFile_doSpace(self) > 0 )
label189:  ¬( self ∈ dom( CreateFile_openFileStore ) )
THEN
label190: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf15   ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label194: self ∈ CreateFile
label195: self ∈ dom(CreateFile_state)
label196: CreateFile_state(self) = cf15
```

```
label197: self ∈ dom( CreateFile_doStore )
label203: target = CreateFile_doStore(self)
THEN
label204: DOStore_freeSpace ( target ) :=
            DOStore_freeSpace ( target ) + 1
label205: CreateFile_state(self) := cf16
END

CreateFile_cf15_isNull   ≙
WHICH IS
ordinary
ANY
self
WHERE
label198: self ∈ CreateFile
label199: self ∈ dom(CreateFile_state)
label200: CreateFile_state(self) = cf15
label201:  ¬( self ∈ dom( CreateFile_doStore ) )
THEN
label202: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf16   ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label206: self ∈ CreateFile
label207: self ∈ dom(CreateFile_state)
label208: CreateFile_state(self) = cf16
label209: self ∈ dom( CreateFile_errorLog )
label215: target = CreateFile_errorLog(self)
label219: ErrorLog_size ( target ) ≥ 0
label220: ErrorLog_size ( target ) < 5
THEN
label216: ErrorLog_error ( target ) :=
            ErrorLog_error ( target ) ⩤
              { ErrorLog_size ( target ) ↦ 6 }
label217: ErrorLog_size ( target ) := ErrorLog_size ( target ) + 1
```

```
label218: ErrorLog_lastIndex ( target ) := ErrorLog_size ( target )
label221: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf16_isNull   ≙
WHICH IS
ordinary
ANY
self
WHERE
label210: self ∈ CreateFile
label211: self ∈ dom(CreateFile_state)
label212: CreateFile_state(self) = cf16
label213:  ¬( self ∈ dom( CreateFile_errorLog ) )
THEN
label214: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf9_else   ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label222: self ∈ CreateFile
label223: self ∈ dom(CreateFile_state)
label224: CreateFile_state(self) = cf9
label225:  ¬(CreateFile_openflSpace(self) > 0 )
label226: self ∈ dom( CreateFile_openFileStore )
label233: target = CreateFile_openFileStore(self)
THEN
label234: OpenFileStore_freeSpace ( target ) :=
            OpenFileStore_freeSpace ( target ) + 1
label235: CreateFile_state(self) := cf17
END

CreateFile_cf9_else_isNull   ≙
WHICH IS
ordinary
ANY
```

```
self
WHERE
label227: self ∈ CreateFile
label228: self ∈ dom(CreateFile_state)
label229: CreateFile_state(self) = cf9
label230:  ¬(CreateFile_openflSpace(self) > 0 )
label231:  ¬( self ∈ dom( CreateFile_openFileStore ) )
THEN
label232: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf17   ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label236: self ∈ CreateFile
label237: self ∈ dom(CreateFile_state)
label238: CreateFile_state(self) = cf17
label239: self ∈ dom( CreateFile_errorLog )
label245: target = CreateFile_errorLog(self)
label249: ErrorLog_size ( target ) ≥ 0
label250: ErrorLog_size ( target ) < 5
THEN
label246: ErrorLog_error ( target ) :=
            ErrorLog_error ( target ) ⩤
              { ErrorLog_size ( target ) ↦ 5 }
label247: ErrorLog_size ( target ) := ErrorLog_size ( target ) + 1
label248: ErrorLog_lastIndex ( target ) := ErrorLog_size ( target )
label251: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf17_isNull   ≙
WHICH IS
ordinary
ANY
self
WHERE
label240: self ∈ CreateFile
```

```
label241: self ∈ dom(CreateFile_state)
label242: CreateFile_state(self) = cf17
label243:  ¬( self ∈ dom( CreateFile_errorLog ) )
THEN
label244: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf7_elseif_0   ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label252: self ∈ CreateFile
label253: self ∈ dom(CreateFile_state)
label254: CreateFile_state(self) = cf7
label255:  ¬( CreateFile_oMode(self) = 1  ∧
            CreateFile_idFound(self) = FALSE  )
label256: CreateFile_oMode(self) ≠ 1
label257: self ∈ dom( CreateFile_errorLog )
label265: target = CreateFile_errorLog(self)
label269: ErrorLog_size ( target ) ≥ 0
label270: ErrorLog_size ( target ) < 5
THEN
label266: ErrorLog_error ( target ) :=
            ErrorLog_error ( target ) ⩤
              { ErrorLog_size ( target ) ↦ 3 }
label267: ErrorLog_size ( target ) := ErrorLog_size ( target ) + 1
label268: ErrorLog_lastIndex ( target ) := ErrorLog_size ( target )
label271: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf7_elseif_0_isNull   ≙
WHICH IS
ordinary
ANY
self
WHERE
label258: self ∈ CreateFile
label259: self ∈ dom(CreateFile_state)
```

```
label260: CreateFile_state(self) = cf7
label261:  ¬( CreateFile_oMode(self) = 1  ∧
            CreateFile_idFound(self) = FALSE  )
label262: CreateFile_oMode(self) ≠ 1
label263:  ¬( self ∈ dom( CreateFile_errorLog ) )
THEN
label264: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf7_elseif_1   ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label272: self ∈ CreateFile
label273: self ∈ dom(CreateFile_state)
label274: CreateFile_state(self) = cf7
label275:  ¬( CreateFile_oMode(self) = 1  ∧
            CreateFile_idFound(self) = FALSE  ∧
            CreateFile_oMode(self) ≠ 1  )
label276: CreateFile_idFound(self) = TRUE
label277: self ∈ dom( CreateFile_errorLog )
label285: target = CreateFile_errorLog(self)
label289: ErrorLog_size ( target ) ≥ 0
label290: ErrorLog_size ( target ) < 5
THEN
label286: ErrorLog_error ( target ) :=
            ErrorLog_error ( target ) ⩤
              { ErrorLog_size ( target ) ↦ 2 }
label287: ErrorLog_size ( target ) := ErrorLog_size ( target ) + 1
label288: ErrorLog_lastIndex ( target ) := ErrorLog_size ( target )
label291: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_cf7_elseif_1_isNull   ≙
WHICH IS
ordinary
ANY
self
```

```
WHERE
label278: self ∈ CreateFile
label279: self ∈ dom(CreateFile_state)
label280: CreateFile_state(self) = cf7
label281:  ¬( CreateFile_oMode(self) = 1  ∧
            CreateFile_idFound(self) = FALSE  ∧
            CreateFile_oMode(self) ≠ 1  )
label282: CreateFile_idFound(self) = TRUE
label283:  ¬( self ∈ dom( CreateFile_errorLog ) )
THEN
label284: CreateFile_state(self) := terminatedCreateFile
END

CreateFile_defaultElse_cf7  ≙
WHICH IS
ordinary
ANY
self
WHERE
label292: self ∈ CreateFile
label293: self ∈ dom(CreateFile_state)
label294: CreateFile_state(self) = cf7
label295:  ¬( CreateFile_oMode(self) = 1  ∧
 CreateFile_idFound(self) = FALSE )
THEN
label299: CreateFile_state(self) := terminatedCreateFile
END

WriteFile_wf1  ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label300: self ∈ WriteFile
label301: self ∈ dom(WriteFile_state)
label302: WriteFile_state(self) = wf1
label303: self ∈ dom( WriteFile_openFileStore )
label309: target = WriteFile_openFileStore(self)
THEN
```

```
label310: WriteFile_openFileCnt ( self ) := OpenFileStore_size ( target )
label311: WriteFile_state(self) := wf2
END

WriteFile_wf1_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label304: self ∈ WriteFile
label305: self ∈ dom(WriteFile_state)
label306: WriteFile_state(self) = wf1
label307:  ¬( self ∈ dom( WriteFile_openFileStore ) )
THEN
label308: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_while_wf2 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label312: self ∈ WriteFile
label313: self ∈ dom(WriteFile_state)
label314: WriteFile_state(self) = wf2
label315: WriteFile_index(self) < WriteFile_openFileCnt(self) ∧
            WriteFile_fileFound(self) = FALSE
label316: self ∈ dom( WriteFile_openFileStore )
label323: target = WriteFile_openFileStore(self)
label325: target ∈ dom( OpenFileStore_openArray )
label326: WriteFile_index ( self ) ≥ 0
label327: WriteFile_index ( self ) < OpenFileStore_size ( target )
label328: WriteFile_index ( self ) ∈
            ( OpenFileStore_openArray ( target ) )
THEN
label324: WriteFile_file ( self ) :=
            OpenFileStore_openArray ( target ) ( WriteFile_index ( self ) )
label329: WriteFile_state(self) := wf3
```

```
END

WriteFile_while_wf2_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label317: self ∈ WriteFile
label318: self ∈ dom(WriteFile_state)
label319: WriteFile_state(self) = wf2
label320: WriteFile_index(self) < WriteFile_openFileCnt(self) ∧ WriteFile_fileFound(s
label321:  ¬( self ∈ dom( WriteFile_openFileStore ) )
THEN
label322: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf3 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label330: self ∈ WriteFile
label331: self ∈ dom(WriteFile_state)
label332: WriteFile_state(self) = wf3
label333: self ∈ dom( WriteFile_file )
label339: target = WriteFile_file(self)
label341: target ∈ dom( OpenFileInfo_dataObject )
THEN
label340: WriteFile_dataObject ( self ) :=
            OpenFileInfo_dataObject ( target )
label342: WriteFile_state(self) := wf4
END

WriteFile_wf3_isNull ≙
WHICH IS
ordinary
ANY
self
```

```
WHERE
label334: self ∈ WriteFile
label335: self ∈ dom(WriteFile_state)
label336: WriteFile_state(self) = wf3
label337:  ¬( self ∈ dom( WriteFile_file ) )
THEN
label338: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf4 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label343: self ∈ WriteFile
label344: self ∈ dom(WriteFile_state)
label345: WriteFile_state(self) = wf4
label346: self ∈ dom( WriteFile_dataObject )
label352: target = WriteFile_dataObject(self)
label354: target ∈ dom( DataObject_fileDirInfo )
THEN
label353: WriteFile_fileDirInfo ( self ) :=
            DataObject_fileDirInfo ( target )
label355: WriteFile_state(self) := wf5
END

WriteFile_wf4_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label347: self ∈ WriteFile
label348: self ∈ dom(WriteFile_state)
label349: WriteFile_state(self) = wf4
label350:  ¬( self ∈ dom( WriteFile_dataObject ) )
THEN
label351: WriteFile_state(self) := terminatedWriteFile
END
```

```
WriteFile_wf5 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label356: self ∈ WriteFile
label357: self ∈ dom(WriteFile_state)
label358: WriteFile_state(self) = wf5
label359: self ∈ dom( WriteFile_fileDirInfo )
label365: target = WriteFile_fileDirInfo(self)
THEN
label366: WriteFile_tmpName ( self ) := FileDirInfo_id ( target )
label367: WriteFile_state(self) := wf6
END

WriteFile_wf5_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label360: self ∈ WriteFile
label361: self ∈ dom(WriteFile_state)
label362: WriteFile_state(self) = wf5
label363:  ¬( self ∈ dom( WriteFile_fileDirInfo ) )
THEN
label364: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf6 ≙
WHICH IS
ordinary
ANY
self
WHERE
label368: self ∈ WriteFile
label369: self ∈ dom(WriteFile_state)
label370: WriteFile_state(self) = wf6
```

```
label373: WriteFile_tmpName(self) = WriteFile_id(self)
THEN
label371: WriteFile_fileFound ( self ) := TRUE
label372: WriteFile_state(self) := wf7
END

WriteFile_defaultElse_wf6 ≙
WHICH IS
ordinary
ANY
self
WHERE
label374: self ∈ WriteFile
label375: self ∈ dom(WriteFile_state)
label376: WriteFile_state(self) = wf6
label377:  ¬( WriteFile_tmpName(self) = WriteFile_id(self)  )
THEN
label378: WriteFile_state(self) := wf7
END

WriteFile_wf7 ≙
WHICH IS
ordinary
ANY
self
WHERE
label379: self ∈ WriteFile
label380: self ∈ dom(WriteFile_state)
label381: WriteFile_state(self) = wf7
THEN
label382: WriteFile_index ( self ) := WriteFile_index ( self ) + 1
label383: WriteFile_state(self) := wf2
END

WriteFile_while_wf2_false ≙
WHICH IS
ordinary
ANY
self
WHERE
label384: self ∈ WriteFile
```

```
label385: self ∈ dom(WriteFile_state)
label386: WriteFile_state(self) = wf2
label387:  ¬( WriteFile_index(self) < WriteFile_openFileCnt(self) ∧
              WriteFile_fileFound(self) = FALSE  )
THEN
label388: WriteFile_state(self) := wf8
END

WriteFile_wf8 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label389: self ∈ WriteFile
label390: self ∈ dom(WriteFile_state)
label391: WriteFile_state(self) = wf8
label392: WriteFile_fileFound(self) = TRUE
label393: self ∈ dom( WriteFile_file )
label400: target = WriteFile_file(self)
THEN
label401: WriteFile_aMode ( self ) := OpenFileInfo_accessMode ( target )
label402: WriteFile_state(self) := wf9
END

WriteFile_wf8_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label394: self ∈ WriteFile
label395: self ∈ dom(WriteFile_state)
label396: WriteFile_state(self) = wf8
label397: WriteFile_fileFound(self) = TRUE
label398:  ¬( self ∈ dom( WriteFile_file ) )
THEN
label399: WriteFile_state(self) := terminatedWriteFile
END
```

```
WriteFile_wf9 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label403: self ∈ WriteFile
label404: self ∈ dom(WriteFile_state)
label405: WriteFile_state(self) = wf9
label406: WriteFile_aMode(self) = 1 ∨ WriteFile_aMode(self) = 2
label407: self ∈ dom( WriteFile_dataObject )
label414: target = WriteFile_dataObject(self)
THEN
label415: WriteFile_freeSpace ( self ) :=
            DataObject_freeSpace ( target )
label416: DataObject_freeSpace ( target ) :=
            DataObject_freeSpace ( target ) - 1
label417: WriteFile_state(self) := wf10
END

WriteFile_wf9_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label408: self ∈ WriteFile
label409: self ∈ dom(WriteFile_state)
label410: WriteFile_state(self) = wf9
label411: WriteFile_aMode(self) = 1 ∨ WriteFile_aMode(self) = 2
label412:  ¬( self ∈ dom( WriteFile_dataObject ) )
THEN
label413: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf10 ≙
WHICH IS
ordinary
ANY
self
```

```
WHERE
label418: self ∈ WriteFile
label419: self ∈ dom(WriteFile_state)
label420: WriteFile_state(self) = wf10
label423: WriteFile_freeSpace(self) > 0
THEN
label421: WriteFile_index ( self ) := 0
label422: WriteFile_state(self) := wf11
END

WriteFile_wf11 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label424: self ∈ WriteFile
label425: self ∈ dom(WriteFile_state)
label426: WriteFile_state(self) = wf11
label427: self ∈ dom( WriteFile_file )
label433: target = WriteFile_file(self)
THEN
label434: OpenFileInfo_fileOffset ( target ) := 0
label435: WriteFile_state(self) := wf12
END

WriteFile_wf11_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label428: self ∈ WriteFile
label429: self ∈ dom(WriteFile_state)
label430: WriteFile_state(self) = wf11
label431:  ¬( self ∈ dom( WriteFile_file ) )
THEN
label432: WriteFile_state(self) := terminatedWriteFile
END
```

```
WriteFile_while_wf12 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label436: self ∈ WriteFile
label437: self ∈ dom(WriteFile_state)
label438: WriteFile_state(self) = wf12
label439: WriteFile_index(self) < WriteFile_bytes(self)
label440: self ∈ dom( WriteFile_buffer )
label447: target = WriteFile_buffer(self)
label449: WriteFile_index ( self ) ≥ 0
label450: WriteFile_index ( self ) < UserBuffer_capacity ( target )
label451: UserBuffer_capacity ( target ) = 10
THEN
label448: WriteFile_data ( self ) :=
            UserBuffer_buffer ( target ) ( WriteFile_index ( self ) )
label452: WriteFile_state(self) := wf13
END

WriteFile_while_wf12_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label441: self ∈ WriteFile
label442: self ∈ dom(WriteFile_state)
label443: WriteFile_state(self) = wf12
label444: WriteFile_index(self) < WriteFile_bytes(self)
label445:  ¬( self ∈ dom( WriteFile_buffer ) )
THEN
label446: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf13 ≙
WHICH IS
ordinary
ANY
```

```
self
target
WHERE
label453: self ∈ WriteFile
label454: self ∈ dom(WriteFile_state)
label455: WriteFile_state(self) = wf13
label456: self ∈ dom( WriteFile_file )
label462: target = WriteFile_file(self)
THEN
label463: WriteFile_offset ( self ) := OpenFileInfo_fileOffset ( target )
label464: WriteFile_state(self) := wf14
END

WriteFile_wf13_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label457: self ∈ WriteFile
label458: self ∈ dom(WriteFile_state)
label459: WriteFile_state(self) = wf13
label460:  ¬( self ∈ dom( WriteFile_file ) )
THEN
label461: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf14 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label465: self ∈ WriteFile
label466: self ∈ dom(WriteFile_state)
label467: WriteFile_state(self) = wf14
label468: self ∈ dom( WriteFile_dataObject )
label474: target = WriteFile_dataObject(self)
label476: WriteFile_offset ( self ) ≥ 0
label477: WriteFile_offset ( self ) < 10
```

```
THEN
label475: DataObject_data ( target ) :=
            DataObject_data ( target ) ⩤
              { WriteFile_offset ( self ) ↦ WriteFile_data ( self ) }
label478: WriteFile_state(self) := wf15
END

WriteFile_wf14_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label469: self ∈ WriteFile
label470: self ∈ dom(WriteFile_state)
label471: WriteFile_state(self) = wf14
label472:  ¬( self ∈ dom( WriteFile_dataObject ) )
THEN
label473: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf15 ≙
WHICH IS
ordinary
ANY
self
WHERE
label479: self ∈ WriteFile
label480: self ∈ dom(WriteFile_state)
label481: WriteFile_state(self) = wf15
THEN
label482: WriteFile_index ( self ) := WriteFile_index ( self ) + 1
label483: WriteFile_state(self) := wf16
END

WriteFile_wf16 ≙
WHICH IS
ordinary
ANY
self
target
```

```
WHERE
label484: self ∈ WriteFile
label485: self ∈ dom(WriteFile_state)
label486: WriteFile_state(self) = wf16
label487: self ∈ dom( WriteFile_file )
label493: target = WriteFile_file(self)
THEN
label494: OpenFileInfo_fileOffset ( target ) :=
            OpenFileInfo_fileOffset ( target ) + 1
label495: WriteFile_state(self) := wf12
END

WriteFile_wf16_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label488: self ∈ WriteFile
label489: self ∈ dom(WriteFile_state)
label490: WriteFile_state(self) = wf16
label491:  ¬( self ∈ dom( WriteFile_file ) )
THEN
label492: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_while_wf12_false ≙
WHICH IS
ordinary
ANY
self
WHERE
label496: self ∈ WriteFile
label497: self ∈ dom(WriteFile_state)
label498: WriteFile_state(self) = wf12
label499:  ¬( WriteFile_index(self) < WriteFile_bytes(self)  )
THEN
label500: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf10_else ≙
```

```
WHICH IS
ordinary
ANY
self
target
WHERE
label501: self ∈ WriteFile
label502: self ∈ dom(WriteFile_state)
label503: WriteFile_state(self) = wf10
label504:  ¬(WriteFile_freeSpace(self) > 0 )
label505: self ∈ dom( WriteFile_dataObject )
label512: target = WriteFile_dataObject(self)
THEN
label513: DataObject_freeSpace ( target ) :=
            DataObject_freeSpace ( target ) + 1
label514: WriteFile_state(self) := wf17
END

WriteFile_wf10_else_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label506: self ∈ WriteFile
label507: self ∈ dom(WriteFile_state)
label508: WriteFile_state(self) = wf10
label509:  ¬(WriteFile_freeSpace(self) > 0 )
label510:  ¬( self ∈ dom( WriteFile_dataObject ) )
THEN
label511: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf17 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label515: self ∈ WriteFile
```

```
label516: self ∈ dom(WriteFile_state)
label517: WriteFile_state(self) = wf17
label518: self ∈ dom( WriteFile_errorLog )
label524: target = WriteFile_errorLog(self)
label528: ErrorLog_size ( target ) ≥ 0
label529: ErrorLog_size ( target ) < 5
THEN
label525: ErrorLog_error ( target ) :=
            ErrorLog_error ( target ) ⩤
              { ErrorLog_size ( target ) ↦ 7 }
label526: ErrorLog_size ( target ) := ErrorLog_size ( target ) + 1
label527: ErrorLog_lastIndex ( target ) := ErrorLog_size ( target )
label530: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf17_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label519: self ∈ WriteFile
label520: self ∈ dom(WriteFile_state)
label521: WriteFile_state(self) = wf17
label522:  ¬( self ∈ dom( WriteFile_errorLog ) )
THEN
label523: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf9_else ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label531: self ∈ WriteFile
label532: self ∈ dom(WriteFile_state)
label533: WriteFile_state(self) = wf9
label534:  ¬(WriteFile_aMode(self) = 1 ∨ WriteFile_aMode(self) = 2 )
label535: self ∈ dom( WriteFile_errorLog )
```

```
label542: target = WriteFile_errorLog(self)
label546: ErrorLog_size ( target ) ≥ 0
label547: ErrorLog_size ( target ) < 5
THEN
label543: ErrorLog_error ( target ) :=
            ErrorLog_error ( target ) ⩤
              { ErrorLog_size ( target ) ↦ 4 }
label544: ErrorLog_size ( target ) := ErrorLog_size ( target ) + 1
label545: ErrorLog_lastIndex ( target ) := ErrorLog_size ( target )
label548: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf9_else_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label536: self ∈ WriteFile
label537: self ∈ dom(WriteFile_state)
label538: WriteFile_state(self) = wf9
label539:  ¬(WriteFile_aMode(self) = 1 ∨ WriteFile_aMode(self) = 2 )
label540:  ¬( self ∈ dom( WriteFile_errorLog ) )
THEN
label541: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf8_else ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label549: self ∈ WriteFile
label550: self ∈ dom(WriteFile_state)
label551: WriteFile_state(self) = wf8
label552:  ¬(WriteFile_fileFound(self) = TRUE )
label553: self ∈ dom( WriteFile_errorLog )
label560: target = WriteFile_errorLog(self)
label564: ErrorLog_size ( target ) ≥ 0
```

```
label565: ErrorLog_size ( target ) < 5
THEN
label561: ErrorLog_error ( target ) :=
            ErrorLog_error ( target ) ⩤
              { ErrorLog_size ( target ) ↦ 1 }
label562: ErrorLog_size ( target ) := ErrorLog_size ( target ) + 1
label563: ErrorLog_lastIndex ( target ) := ErrorLog_size ( target )
label566: WriteFile_state(self) := terminatedWriteFile
END

WriteFile_wf8_else_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label554: self ∈ WriteFile
label555: self ∈ dom(WriteFile_state)
label556: WriteFile_state(self) = wf8
label557:  ¬(WriteFile_fileFound(self) = TRUE )
label558:  ¬( self ∈ dom( WriteFile_errorLog ) )
THEN
label559: WriteFile_state(self) := terminatedWriteFile
END

ReadFile_rf1 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label567: self ∈ ReadFile
label568: self ∈ dom(ReadFile_state)
label569: ReadFile_state(self) = rf1
label570: self ∈ dom( ReadFile_openFileStore )
label576: target = ReadFile_openFileStore(self)
THEN
label577: ReadFile_openFileCnt ( self ) := OpenFileStore_size ( target )
label578: ReadFile_state(self) := rf2
END
```

```
ReadFile_rf1_isNull ≜
WHICH IS
ordinary
ANY
self
WHERE
label571: self ∈ ReadFile
label572: self ∈ dom(ReadFile_state)
label573: ReadFile_state(self) = rf1
label574:  ¬( self ∈ dom( ReadFile_openFileStore ) )
THEN
label575: ReadFile_state(self) := terminatedReadFile
END

ReadFile_while_rf2 ≜
WHICH IS
ordinary
ANY
self
target
WHERE
label579: self ∈ ReadFile
label580: self ∈ dom(ReadFile_state)
label581: ReadFile_state(self) = rf2
label582: ReadFile_index(self) < ReadFile_openFileCnt(self) ∧
            ReadFile_fileFound(self) = FALSE
label583: self ∈ dom( ReadFile_openFileStore )
label590: target = ReadFile_openFileStore(self)
label592: target ∈ dom( OpenFileStore_openArray )
label593: ReadFile_index ( self ) ≥ 0
label594: ReadFile_index ( self ) < OpenFileStore_size ( target )
label595: ReadFile_index ( self ) ∈
            ( OpenFileStore_openArray ( target ) )
THEN
label591: ReadFile_file ( self ) :=
            OpenFileStore_openArray ( target ) ( ReadFile_index ( self ) )
label596: ReadFile_state(self) := rf3
END

ReadFile_while_rf2_isNull ≜
```

```
WHICH IS
ordinary
ANY
self
WHERE
label584: self ∈ ReadFile
label585: self ∈ dom(ReadFile_state)
label586: ReadFile_state(self) = rf2
label587: ReadFile_index(self) < ReadFile_openFileCnt(self) ∧
            ReadFile_fileFound(self) = FALSE
label588:  ¬( self ∈ dom( ReadFile_openFileStore ) )
THEN
label589: ReadFile_state(self) := terminatedReadFile
END

ReadFile_rf3 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label597: self ∈ ReadFile
label598: self ∈ dom(ReadFile_state)
label599: ReadFile_state(self) = rf3
label600: self ∈ dom( ReadFile_file )
label606: target = ReadFile_file(self)
label608: target ∈ dom( OpenFileInfo_dataObject )
THEN
label607: ReadFile_dataObject ( self ) :=
            OpenFileInfo_dataObject ( target )
label609: ReadFile_state(self) := rf4
END

ReadFile_rf3_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label601: self ∈ ReadFile
```

```
label602: self ∈ dom(ReadFile_state)
label603: ReadFile_state(self) = rf3
label604:  ¬( self ∈ dom( ReadFile_file ) )
THEN
label605: ReadFile_state(self) := terminatedReadFile
END

ReadFile_rf4 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label610: self ∈ ReadFile
label611: self ∈ dom(ReadFile_state)
label612: ReadFile_state(self) = rf4
label613: self ∈ dom( ReadFile_dataObject )
label619: target = ReadFile_dataObject(self)
label621: target ∈ dom( DataObject_fileDirInfo )
THEN
label620: ReadFile_fileDirInfo ( self ) :=
            DataObject_fileDirInfo ( target )
label622: ReadFile_state(self) := rf5
END

ReadFile_rf4_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label614: self ∈ ReadFile
label615: self ∈ dom(ReadFile_state)
label616: ReadFile_state(self) = rf4
label617:  ¬( self ∈ dom( ReadFile_dataObject ) )
THEN
label618: ReadFile_state(self) := terminatedReadFile
END

ReadFile_rf5 ≙
```

```
WHICH IS
ordinary
ANY
self
target
WHERE
label623: self ∈ ReadFile
label624: self ∈ dom(ReadFile_state)
label625: ReadFile_state(self) = rf5
label626: self ∈ dom( ReadFile_fileDirInfo )
label632: target = ReadFile_fileDirInfo(self)
THEN
label633: ReadFile_tmpName ( self ) := FileDirInfo_id ( target )
label634: ReadFile_state(self) := rf6
END

ReadFile_rf5_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label627: self ∈ ReadFile
label628: self ∈ dom(ReadFile_state)
label629: ReadFile_state(self) = rf5
label630:  ¬( self ∈ dom( ReadFile_fileDirInfo ) )
THEN
label631: ReadFile_state(self) := terminatedReadFile
END

ReadFile_rf6 ≙
WHICH IS
ordinary
ANY
self
WHERE
label635: self ∈ ReadFile
label636: self ∈ dom(ReadFile_state)
label637: ReadFile_state(self) = rf6
label640: ReadFile_tmpName(self) = ReadFile_id(self)
THEN
```

```
label638: ReadFile_fileFound ( self ) := TRUE
label639: ReadFile_state(self) := rf7
END

ReadFile_defaultElse_rf6 ≙
WHICH IS
ordinary
ANY
self
WHERE
label641: self ∈ ReadFile
label642: self ∈ dom(ReadFile_state)
label643: ReadFile_state(self) = rf6
label644:  ¬( ReadFile_tmpName(self) = ReadFile_id(self)  )
THEN
label645: ReadFile_state(self) := rf7
END

ReadFile_rf7 ≙
WHICH IS
ordinary
ANY
self
WHERE
label646: self ∈ ReadFile
label647: self ∈ dom(ReadFile_state)
label648: ReadFile_state(self) = rf7
THEN
label649: ReadFile_index ( self ) := ReadFile_index ( self ) + 1
label650: ReadFile_state(self) := rf2
END

ReadFile_while_rf2_false ≙
WHICH IS
ordinary
ANY
self
WHERE
label651: self ∈ ReadFile
label652: self ∈ dom(ReadFile_state)
label653: ReadFile_state(self) = rf2
```

```
label654:  ¬( ReadFile_index(self) < ReadFile_openFileCnt(self) ∧
            ReadFile_fileFound(self) = FALSE  )
THEN
label655: ReadFile_state(self) := rf8
END

ReadFile_rf8 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label656: self ∈ ReadFile
label657: self ∈ dom(ReadFile_state)
label658: ReadFile_state(self) = rf8
label659: ReadFile_fileFound(self) = TRUE
label660: self ∈ dom( ReadFile_file )
label667: target = ReadFile_file(self)
THEN
label668: ReadFile_aMode ( self ) := OpenFileInfo_accessMode ( target )
label669: ReadFile_state(self) := rf9
END

ReadFile_rf8_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label661: self ∈ ReadFile
label662: self ∈ dom(ReadFile_state)
label663: ReadFile_state(self) = rf8
label664: ReadFile_fileFound(self) = TRUE
label665:  ¬( self ∈ dom( ReadFile_file ) )
THEN
label666: ReadFile_state(self) := terminatedReadFile
END

ReadFile_rf9 ≙
WHICH IS
```

```
ordinary
ANY
self
WHERE
label670: self ∈ ReadFile
label671: self ∈ dom(ReadFile_state)
label672: ReadFile_state(self) = rf9
label675: ReadFile_aMode(self) = 0 ∨ ReadFile_aMode(self) = 2
THEN
label673: ReadFile_index ( self ) := 0
label674: ReadFile_state(self) := rf10
END

ReadFile_rf10 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label676: self ∈ ReadFile
label677: self ∈ dom(ReadFile_state)
label678: ReadFile_state(self) = rf10
label679: self ∈ dom( ReadFile_file )
label685: target = ReadFile_file(self)
THEN
label686: OpenFileInfo_fileOffset ( target ) := 0
label687: ReadFile_state(self) := rf11
END

ReadFile_rf10_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label680: self ∈ ReadFile
label681: self ∈ dom(ReadFile_state)
label682: ReadFile_state(self) = rf10
label683:  ¬( self ∈ dom( ReadFile_file ) )
THEN
```

```
label684: ReadFile_state(self) := terminatedReadFile
END

ReadFile_while_rf11 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label688: self ∈ ReadFile
label689: self ∈ dom(ReadFile_state)
label690: ReadFile_state(self) = rf11
label691: ReadFile_index(self) < ReadFile_bytes(self)
label692: self ∈ dom( ReadFile_file )
label699: target = ReadFile_file(self)
THEN
label700: ReadFile_offset ( self ) := OpenFileInfo_fileOffset ( target )
label701: ReadFile_state(self) := rf12
END

ReadFile_while_rf11_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label693: self ∈ ReadFile
label694: self ∈ dom(ReadFile_state)
label695: ReadFile_state(self) = rf11
label696: ReadFile_index(self) < ReadFile_bytes(self)
label697:  ¬( self ∈ dom( ReadFile_file ) )
THEN
label698: ReadFile_state(self) := terminatedReadFile
END

ReadFile_rf12 ≙
WHICH IS
ordinary
ANY
self
```

```
target
WHERE
label702: self ∈ ReadFile
label703: self ∈ dom(ReadFile_state)
label704: ReadFile_state(self) = rf12
label705: self ∈ dom( ReadFile_dataObject )
label711: target = ReadFile_dataObject(self)
label713: ReadFile_offset ( self ) ≥ 0
label714: ReadFile_offset ( self ) < 10
THEN
label712: ReadFile_data ( self ) :=
            DataObject_data ( target ) ( ReadFile_offset ( self ) )
label715: ReadFile_state(self) := rf13
END

ReadFile_rf12_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label706: self ∈ ReadFile
label707: self ∈ dom(ReadFile_state)
label708: ReadFile_state(self) = rf12
label709:  ¬( self ∈ dom( ReadFile_dataObject ) )
THEN
label710: ReadFile_state(self) := terminatedReadFile
END

ReadFile_rf13 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label716: self ∈ ReadFile
label717: self ∈ dom(ReadFile_state)
label718: ReadFile_state(self) = rf13
label719: self ∈ dom( ReadFile_buffer )
label725: target = ReadFile_buffer(self)
```

```
label728: UserBuffer_size ( target ) ≥ 0
label729: UserBuffer_size ( target ) < UserBuffer_capacity ( target )
label730: UserBuffer_capacity ( target ) = 10
THEN
label726: UserBuffer_buffer ( target ) :=
            UserBuffer_buffer ( target ) ⩤
              { UserBuffer_size ( target ) ↦ ReadFile_data ( self ) }
label727: UserBuffer_size ( target ) := UserBuffer_size ( target ) + 1
label731: ReadFile_state(self) := rf14
END

ReadFile_rf13_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label720: self ∈ ReadFile
label721: self ∈ dom(ReadFile_state)
label722: ReadFile_state(self) = rf13
label723:  ¬( self ∈ dom( ReadFile_buffer ) )
THEN
label724: ReadFile_state(self) := terminatedReadFile
END

ReadFile_rf14 ≙
WHICH IS
ordinary
ANY
self
WHERE
label732: self ∈ ReadFile
label733: self ∈ dom(ReadFile_state)
label734: ReadFile_state(self) = rf14
THEN
label735: ReadFile_index ( self ) := ReadFile_index ( self ) + 1
label736: ReadFile_state(self) := rf15
END

ReadFile_rf15 ≙
WHICH IS
```

```
ordinary
ANY
self
target
WHERE
label737: self ∈ ReadFile
label738: self ∈ dom(ReadFile_state)
label739: ReadFile_state(self) = rf15
label740: self ∈ dom( ReadFile_file )
label746: target = ReadFile_file(self)
THEN
label747: OpenFileInfo_fileOffset ( target ) :=
            OpenFileInfo_fileOffset ( target ) + 1
label748: ReadFile_state(self) := rf11
END

ReadFile_rf15_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label741: self ∈ ReadFile
label742: self ∈ dom(ReadFile_state)
label743: ReadFile_state(self) = rf15
label744:  ¬( self ∈ dom( ReadFile_file ) )
THEN
label745: ReadFile_state(self) := terminatedReadFile
END

ReadFile_while_rf11_false ≙
WHICH IS
ordinary
ANY
self
WHERE
label749: self ∈ ReadFile
label750: self ∈ dom(ReadFile_state)
label751: ReadFile_state(self) = rf11
label752:  ¬( ReadFile_index(self) < ReadFile_bytes(self)  )
THEN
```

```
label753: ReadFile_state(self) := terminatedReadFile
END

ReadFile_rf9_else ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label754: self ∈ ReadFile
label755: self ∈ dom(ReadFile_state)
label756: ReadFile_state(self) = rf9
label757:  ¬(ReadFile_aMode(self) = 0 ∨ ReadFile_aMode(self) = 2 )
label758: self ∈ dom( ReadFile_errorLog )
label765: target = ReadFile_errorLog(self)
label769: ErrorLog_size ( target ) ≥ 0
label770: ErrorLog_size ( target ) < 5
THEN
label766: ErrorLog_error ( target ) :=
            ErrorLog_error ( target ) ⩤
              { ErrorLog_size ( target ) ↦ 4 }
label767: ErrorLog_size ( target ) := ErrorLog_size ( target ) + 1
label768: ErrorLog_lastIndex ( target ) := ErrorLog_size ( target )
label771: ReadFile_state(self) := terminatedReadFile
END

ReadFile_rf9_else_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label759: self ∈ ReadFile
label760: self ∈ dom(ReadFile_state)
label761: ReadFile_state(self) = rf9
label762:  ¬(ReadFile_aMode(self) = 0 ∨ ReadFile_aMode(self) = 2 )
label763:  ¬( self ∈ dom( ReadFile_errorLog ) )
THEN
label764: ReadFile_state(self) := terminatedReadFile
END
```

```
ReadFile_rf8_else ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label772: self ∈ ReadFile
label773: self ∈ dom(ReadFile_state)
label774: ReadFile_state(self) = rf8
label775:  ¬(ReadFile_fileFound(self) = TRUE )
label776: self ∈ dom( ReadFile_errorLog )
label783: target = ReadFile_errorLog(self)
label787: ErrorLog_size ( target ) ≥ 0
label788: ErrorLog_size ( target ) < 5
THEN
label784: ErrorLog_error ( target ) :=
            ErrorLog_error ( target ) ⩤
              { ErrorLog_size ( target ) ↦ 1 }
label785: ErrorLog_size ( target ) := ErrorLog_size ( target ) + 1
label786: ErrorLog_lastIndex ( target ) := ErrorLog_size ( target )
label789: ReadFile_state(self) := terminatedReadFile
END

ReadFile_rf8_else_isNull ≙
WHICH IS
ordinary
ANY
self
WHERE
label777: self ∈ ReadFile
label778: self ∈ dom(ReadFile_state)
label779: ReadFile_state(self) = rf8
label780:  ¬(ReadFile_fileFound(self) = TRUE )
label781:  ¬( self ∈ dom( ReadFile_errorLog ) )
THEN
label782: ReadFile_state(self) := terminatedReadFile
END

UserAppCreateFile_ua1 ≙
```

```
WHICH IS
ordinary
ANY
self
new
WHERE
label790: self ∈ UserAppCreateFile
label791: self ∈ dom(UserAppCreateFile_state)
label792: new ∈ CreateFile_Set \ CreateFile
label793: UserAppCreateFile_state(self) = ua1
label794: self ∈ dom(UserAppCreateFile_doStore)
label795: self ∈ dom(UserAppCreateFile_openFileStore)
label796: self ∈ dom(UserAppCreateFile_errorLog)
THEN
label797: CreateFile_doSpace ( new ) := - 1
label798: CreateFile_openflSpace ( new ) := - 1
label799: CreateFile_doStore ( new ) :=
            UserAppCreateFile_doStore ( self )
label800: CreateFile_openFileStore ( new ) :=
            UserAppCreateFile_openFileStore ( self )
label801: CreateFile_id ( new ) := 0
label802: CreateFile_oMode ( new ) := 1
label803: CreateFile_sMode ( new ) := 0
label804: CreateFile_aMode ( new ) := - 1
label805: CreateFile_tmpName ( new ) := - 1
label806: CreateFile_idFound ( new ) := FALSE
label807: CreateFile_index ( new ) := 0
label808: CreateFile_errorLog ( new ) :=
            UserAppCreateFile_errorLog ( self )
label809: CreateFile_state(new):=cf1
label810: CreateFile := CreateFile ∪ {new}
label811: UserAppCreateFile_createFile(self) := new
label812: UserAppCreateFile_state(self) := terminatedUserAppCreateFile
END

UserAppWriteFile_uaw1 ≙
WHICH IS
ordinary
ANY
self
new
```

```
WHERE
label813: self ∈ UserAppWriteFile
label814: self ∈ dom(UserAppWriteFile_state)
label815: new ∈ UserBuffer_Set \ UserBuffer
label816: UserAppWriteFile_state(self) = uaw1
THEN
label817: UserBuffer_buffer(new):= λ i· i ∈ 0 .. 9|0
label818: UserBuffer_capacity ( new ) := 10
label819: UserBuffer_size ( new ) := 0
label820: UserBuffer := UserBuffer ∪ {new}
label821: UserAppWriteFile_buff1(self) := new
label822: UserAppWriteFile_state(self) := uaw2
END

UserAppWriteFile_while_uaw2 ≙
WHICH IS
ordinary
ANY
self
target
WHERE
label823: self ∈ UserAppWriteFile
label824: self ∈ dom(UserAppWriteFile_state)
label825: UserAppWriteFile_state(self) = uaw2
label826: UserAppWriteFile_data(self) < 70
label827: self ∈ dom( UserAppWriteFile_buff1 )
label834: target = UserAppWriteFile_buff1(self)
label837: UserBuffer_size ( target ) ≥ 0
label838: UserBuffer_size ( target ) < UserBuffer_capacity ( target )
label839: UserBuffer_capacity ( target ) = 10
THEN
label835: UserBuffer_buffer ( target ) :=
            UserBuffer_buffer ( target ) ⩤
              { UserBuffer_size ( target ) ↦ UserAppWriteFile_data ( self ) }
label836: UserBuffer_size ( target ) := UserBuffer_size ( target ) + 1
label840: UserAppWriteFile_state(self) := uaw3
END

UserAppWriteFile_while_uaw2_isNull ≙
WHICH IS
ordinary
```

```
ANY
self
WHERE
label828: self ∈ UserAppWriteFile
label829: self ∈ dom(UserAppWriteFile_state)
label830: UserAppWriteFile_state(self) = uaw2
label831: UserAppWriteFile_data(self) < 70
label832:  ¬( self ∈ dom( UserAppWriteFile_buff1 ) )
THEN
label833: UserAppWriteFile_state(self) := terminatedUserAppWriteFile
END

UserAppWriteFile_uaw3 ≙
WHICH IS
ordinary
ANY
self
WHERE
label841: self ∈ UserAppWriteFile
label842: self ∈ dom(UserAppWriteFile_state)
label843: UserAppWriteFile_state(self) = uaw3
THEN
label844: UserAppWriteFile_data ( self ) :=
            UserAppWriteFile_data ( self ) + 1
label845: UserAppWriteFile_state(self) := uaw2
END

UserAppWriteFile_while_uaw2_false ≙
WHICH IS
ordinary
ANY
self
WHERE
label846: self ∈ UserAppWriteFile
label847: self ∈ dom(UserAppWriteFile_state)
label848: UserAppWriteFile_state(self) = uaw2
label849:  ¬( UserAppWriteFile_data(self) < 70  )
THEN
label850: UserAppWriteFile_state(self) := uaw4
END
```

```
UserAppWriteFile_uaw4 ≙
WHICH IS
ordinary
ANY
self
new
WHERE
label851: self ∈ UserAppWriteFile
label852: self ∈ dom(UserAppWriteFile_state)
label853: new ∈ WriteFile_Set \ WriteFile
label854: UserAppWriteFile_state(self) = uaw4
label855: self ∈ dom(UserAppWriteFile_openFileStore)
label856: self ∈ dom(UserAppWriteFile_buff1)
label857: self ∈ dom(UserAppWriteFile_errorLog)
THEN
label858: WriteFile_openFileStore ( new ) :=
            UserAppWriteFile_openFileStore ( self )
label859: WriteFile_id ( new ) := 0
label860: WriteFile_buffer ( new ) := UserAppWriteFile_buff1 ( self )
label861: WriteFile_bytes ( new ) := 5
label862: WriteFile_index ( new ) := 0
label863: WriteFile_openFileCnt ( new ) := 0
label864: WriteFile_fileFound ( new ) := FALSE
label865: WriteFile_tmpName ( new ) := - 1
label866: WriteFile_data ( new ) := - 1
label867: WriteFile_offset ( new ) := 0
label868: WriteFile_aMode ( new ) := - 1
label869: WriteFile_errorLog ( new ) :=
            UserAppWriteFile_errorLog ( self )
label870: WriteFile_freeSpace ( new ) := 0
label871: WriteFile_state(new):=wf1
label872: WriteFile := WriteFile ∪ {new}
label873: UserAppWriteFile_writeFile1(self) := new
label874: UserAppWriteFile_state(self) := terminatedUserAppWriteFile
END
UserAppReadFile_uar1 ≙
WHICH IS
ordinary
ANY
self
new
```

```
WHERE
label875: self ∈ UserAppReadFile
label876: self ∈ dom(UserAppReadFile_state)
label877: new ∈ UserBuffer_Set \ UserBuffer
label878: UserAppReadFile_state(self) = uar1
THEN
label879: UserBuffer_buffer(new):= λ i· i ∈ 0 .. 9|0
label880: UserBuffer_capacity ( new ) := 10
label881: UserBuffer_size ( new ) := 0
label882: UserBuffer := UserBuffer ∪ {new}
label883: UserAppReadFile_buff1(self) := new
label884: UserAppReadFile_state(self) := uar2
END

UserAppReadFile_uar2 ≙
WHICH IS
ordinary
ANY
self
new
WHERE
label885: self ∈ UserAppReadFile
label886: self ∈ dom(UserAppReadFile_state)
label887: new ∈ ReadFile_Set \ ReadFile
label888: UserAppReadFile_state(self) = uar2
label889: self ∈ dom(UserAppReadFile_openFileStore)
label890: self ∈ dom(UserAppReadFile_buff1)
label891: self ∈ dom(UserAppReadFile_errorLog)
THEN
label892: ReadFile_openFileStore ( new ) :=
            UserAppReadFile_openFileStore ( self )
label893: ReadFile_id ( new ) := 0
label894: ReadFile_buffer ( new ) := UserAppReadFile_buff1 ( self )
label895: ReadFile_bytes ( new ) := 5
label896: ReadFile_index ( new ) := 0
label897: ReadFile_openFileCnt ( new ) := 0
label898: ReadFile_fileFound ( new ) := FALSE
label899: ReadFile_tmpName ( new ) := - 1
label900: ReadFile_data ( new ) := - 1
label901: ReadFile_offset ( new ) := 0
label902: ReadFile_aMode ( new ) := - 1
```

```
label903: ReadFile_errorLog ( new ) := UserAppReadFile_errorLog ( self )
label904: ReadFile_state(new):=rf1
label905: ReadFile := ReadFile ∪ {new}
label906: UserAppReadFile_readFile1(self) := new
label907: UserAppReadFile_state(self) := terminatedUserAppReadFile
END

loadFFS2 ≙
WHICH IS
ordinary
ANY
self
WHERE
self: self ∈ FFS2_Set \ FFS2
THEN
label908: FFS2 := FFS2 ∪ {self}
label909: FFS2_state(self) := s1
END

FFS2_s1 ≙
WHICH IS
ordinary
ANY
self
new
WHERE
label910: self ∈ FFS2
label911: self ∈ dom(FFS2_state)
label912: new ∈ DOStore_Set \ DOStore
label913: FFS2_state(self) = s1
THEN
label914: DOStore_doArray(new) := ∅
label915: DOStore_size ( new ) := 0
label916: DOStore_capacity ( new ) := 5
label917: DOStore_freeSpace ( new ) := 5
label918: DOStore := DOStore ∪ {new}
label919: FFS2_doStore(self) := new
label920: FFS2_state(self) := s2
END

FFS2_s2 ≙
```

```
WHICH IS
ordinary
ANY
self
new
WHERE
label921: self ∈ FFS2
label922: self ∈ dom(FFS2_state)
label923: new ∈ OpenFileStore_Set \ OpenFileStore
label924: FFS2_state(self) = s2
THEN
label925: OpenFileStore_openArray(new) := ∅
label926: OpenFileStore_size ( new ) := 0
label927: OpenFileStore_capacity ( new ) := 5
label928: OpenFileStore_freeSpace ( new ) := 5
label929: OpenFileStore := OpenFileStore ∪ {new}
label930: FFS2_openFileStore(self) := new
label931: FFS2_state(self) := s3
END

FFS2_s3 ≙
WHICH IS
ordinary
ANY
self
new
WHERE
label932: self ∈ FFS2
label933: self ∈ dom(FFS2_state)
label934: new ∈ ErrorLog_Set \ ErrorLog
label935: FFS2_state(self) = s3
THEN
label936: ErrorLog_error(new):= λ i· i ∈ 0 .. 4|0
label937: ErrorLog_size ( new ) := 0
label938: ErrorLog_lastIndex ( new ) := - 1
label939: ErrorLog := ErrorLog ∪ {new}
label940: FFS2_errorLog(self) := new
label941: FFS2_state(self) := s4
END

FFS2_s4 ≙
```

```
WHICH IS
ordinary
ANY
self
new
WHERE
label942: self ∈ FFS2
label943: self ∈ dom(FFS2_state)
label944: new ∈ UserAppCreateFile_Set \ UserAppCreateFile
label945: FFS2_state(self) = s4
label946: self ∈ dom(FFS2_doStore)
label947: self ∈ dom(FFS2_openFileStore)
label948: self ∈ dom(FFS2_errorLog)
THEN
label949: UserAppCreateFile_doStore ( new ) := FFS2_doStore ( self )
label950: UserAppCreateFile_openFileStore ( new ) :=
            FFS2_openFileStore ( self )
label951: UserAppCreateFile_errorLog ( new ) := FFS2_errorLog ( self )
label952: UserAppCreateFile_state(new):=ua1
label953: UserAppCreateFile := UserAppCreateFile ∪ {new}
label954: FFS2_userAppCreateFile(self) := new
label955: FFS2_state(self) := s5
END

FFS2_s5 ≙
WHICH IS
ordinary
ANY
self
new
WHERE
label956: self ∈ FFS2
label957: self ∈ dom(FFS2_state)
label958: new ∈ UserAppWriteFile_Set \ UserAppWriteFile
label959: FFS2_state(self) = s5
label960: self ∈ dom(FFS2_openFileStore)
label961: self ∈ dom(FFS2_errorLog)
THEN
label962: UserAppWriteFile_openFileStore ( new ) :=
            FFS2_openFileStore ( self )
label963: UserAppWriteFile_data ( new ) := 65
```

```
label964: UserAppWriteFile_errorLog ( new ) := FFS2_errorLog ( self )
label965: UserAppWriteFile_state(new):=uaw1
label966: UserAppWriteFile := UserAppWriteFile ∪ {new}
label967: FFS2_userAppWriteFile(self) := new
label968: FFS2_state(self) := s6
END

FFS2_s6 ≙
WHICH IS
ordinary
ANY
self
new
WHERE
label969: self ∈ FFS2
label970: self ∈ dom(FFS2_state)
label971: new ∈ UserAppReadFile_Set \ UserAppReadFile
label972: FFS2_state(self) = s6
label973: self ∈ dom(FFS2_openFileStore)
label974: self ∈ dom(FFS2_errorLog)
THEN
label975: UserAppReadFile_openFileStore ( new ) :=
            FFS2_openFileStore ( self )
label976: UserAppReadFile_errorLog ( new ) := FFS2_errorLog ( self )
label977: UserAppReadFile_state(new):=uar1
label978: UserAppReadFile := UserAppReadFile ∪ {new}
label979: FFS2_userAppReadFile(self) := new
label980: FFS2_state(self) := terminatedFFS2
END
END
```

D.16 The MainClass Java Code

```
public class FFS2 {

 private static DOStore doStore = null;
 private static OpenFileStore openFileStore = null;
 private static UserAppCreateFile userAppCreateFile = null;
 private static UserAppWriteFile userAppWriteFile = null;
 private static UserAppReadFile userAppReadFile = null;
```

```
 private static ErrorLog errorLog = null;

 public static void main(String[] args) {
  doStore = new DOStore(); /* s1 */
  openFileStore = new OpenFileStore(); /* s2 */
  errorLog = new ErrorLog(); /* s3 */
  userAppCreateFile =
     new UserAppCreateFile(doStore, openFileStore, errorLog);
  new Thread(userAppCreateFile).start(); /* s4 */
  userAppWriteFile = new UserAppWriteFile(openFileStore, errorLog);
  new Thread(userAppWriteFile).start(); /* s5 */
  userAppReadFile = new UserAppReadFile(openFileStore, errorLog);
  new Thread(userAppReadFile).start(); /* s6 */
 }
}
```

D.17 CreateFile Java Code

```
public class CreateFile implements Runnable {

 private DataObject newFile = null;
 private DOStore doStore = null;
 private int doSpace;
 private int openflSpace;
 private OpenFileInfo openFileInfo = null;
 private OpenFileStore openFileStore = null;
 private int id;
 private FileDirInfo fileDirInfo = null;
 private int oMode;
 private int sMode;
 private int aMode;
 private DataObject tmpObj = null;
 private int tmpName;
 private boolean idFound;
 private int index;
 private ErrorLog errorLog = null;

 public CreateFile(DOStore doStor, OpenFileStore openFileStor, int nameID,
    int oMde, int sMde, ErrorLog errorLg) {
  doSpace = -1; openflSpace = -1; doStore = doStor;
```

```
  openFileStore = openFileStor; id = nameID; oMode = oMde;
  sMode = sMde; aMode = -1; tmpName = -1; idFound = false;
  index = 0; errorLog = errorLg;
 }

 public void run() {
  doSpace = doStore.getSize(); /* cf1 */
  while (index < doSpace && idFound == false) {
   tmpObj = doStore.getAtIndex(index); /* cf2 */
   fileDirInfo = tmpObj.getFileDirInfo(); /* cf3 */
   tmpName = fileDirInfo.getID(); /* cf4 */
   if (tmpName == id) {
    idFound = true; /* cf5 */
   }
   index = index + 1; /* cf6 */
  }
  if (oMode == 1 && idFound == false) {
   aMode = 2; /* cf7 */
   openflSpace = openFileStore.reserveSpace(); /* cf8 */
   if (openflSpace > 0) {
    doSpace = doStore.reserveSpace(); /* cf9 */
    if (doSpace > 0) {
     fileDirInfo = new FileDirInfo(id); /* cf10 */
     newFile = new DataObject(128, fileDirInfo); /* cf11 */
     openFileInfo = new OpenFileInfo(aMode, sMode, newFile); /* cf12 */
     doStore.add(newFile); /* cf13 */
     openFileStore.add(openFileInfo); /* cf14 */
    } else {
     openFileStore.unReserve();
     doStore.unReserve(); /* cf15 */
     errorLog.add(6); /* cf16 */
    }
   } else {
    openFileStore.unReserve();
    errorLog.add(5); /* cf17 */
   }
  } else if (oMode != 1) {
   errorLog.add(3);
  } else if (idFound == true) {
   errorLog.add(2);
  }
```

```
 }
}
```

D.18 WriteFile Java Code

```
public class WriteFile implements Runnable {

 private OpenFileStore openFileStore = null;
 private UserBuffer buffer = null;
 private int id;
 private int tmpName;
 private OpenFileInfo file = null;
 private int bytes;
 private int index;
 private int openFileCnt;
 private boolean fileFound;
 private FileDirInfo fileDirInfo = null;
 private int data;
 private DataObject dataObject = null;
 private int offset;
 private int aMode;
 private ErrorLog errorLog = null;
 private int freeSpace;

 public WriteFile(OpenFileStore openFileStor, int fName,
    UserBuffer buffr, int byts, ErrorLog errorLg) {
  openFileStore = openFileStor; id = fName; buffer = buffr;
  bytes = byts; index = 0; openFileCnt = 0; fileFound = false;
  tmpName = -1; data = -1; offset = 0; aMode = -1; errorLog = errorLg;
  freeSpace = 0;
 }

 public void run() {
  openFileCnt = openFileStore.getSize(); /* wf1 */
  while (index < openFileCnt && fileFound == false) {
   file = openFileStore.getAtIndex(index); /* wf2 */
   dataObject = file.getDataObject(); /* wf3 */
   fileDirInfo = dataObject.getFileDirInfo(); /* wf4 */
   tmpName = fileDirInfo.getID(); /* wf5 */
   if (tmpName == id) {
```

```
      fileFound = true; /* wf6 */
    }
    index = index + 1; /* wf7 */
  }
  if (fileFound == true) {
    aMode = file.getAccessMode(); /* wf8 */
    if (aMode == 1 || aMode == 2) {
      freeSpace = dataObject.reserveSpace(); /* wf9 */
      if (freeSpace > 0) {
        index = 0; /* wf10 */
        file.resetOffset(); /* wf11 */
        while (index < bytes) {
          data = buffer.get(index); /* wf12 */
          offset = file.getOffset(); /* wf13 */
          dataObject.write(data, offset); /* wf14 */
          index = index + 1; /* wf15 */
          file.incOffset(); /* wf16 */
        }
      } else {
        dataObject.unReserve();
        errorLog.add(7); /* wf17 */
      }
    } else {
      errorLog.add(4);
    }
  } else {
    errorLog.add(1);
  }
 }
}
```

D.19 ReadFile Java Code

```
public class ReadFile implements Runnable {

 private OpenFileStore openFileStore = null;
 private UserBuffer buffer = null;
 private int id;
 private int tmpName;
 private OpenFileInfo file = null;
```

```
private int bytes;
private int index;
private int openFileCnt;
private boolean fileFound;
private FileDirInfo fileDirInfo = null;
private int data;
private DataObject dataObject = null;
private int offset;
private int aMode;
private ErrorLog errorLog = null;

public ReadFile(OpenFileStore openFileStor, int fName, UserBuffer buffr,
 int byts, ErrorLog errorLg) {
 openFileStore = openFileStor; id = fName; buffer = buffr; bytes = byts;
 index = 0; openFileCnt = 0; fileFound = false; tmpName = -1; data = -1;
 offset = 0; aMode = -1; errorLog = errorLg;
}

public void run() {
 openFileCnt = openFileStore.getSize(); /* rf1 */
 while (index < openFileCnt && fileFound == false) {
  file = openFileStore.getAtIndex(index); /* rf2 */
  dataObject = file.getDataObject(); /* rf3 */
  fileDirInfo = dataObject.getFileDirInfo(); /* rf4 */
  tmpName = fileDirInfo.getID(); /* rf5 */
  if (tmpName == id) {
   fileFound = true; /* rf6 */
  }
  index = index + 1; /* rf7 */
 }
 if (fileFound == true) {
  aMode = file.getAccessMode(); /* rf8 */
  if (aMode == 0 || aMode == 2) {
   index = 0; /* rf9 */
   file.resetOffset(); /* rf10 */
   while (index < bytes) {
    offset = file.getOffset(); /* rf11 */
    data = dataObject.read(offset); /* rf12 */
    buffer.add(data); /* rf13 */
    index = index + 1; /* rf14 */
    file.incOffset(); /* rf15 */
```

```
      }
    } else {
      errorLog.add(4);
    }
  } else {
    errorLog.add(1);
  }
 }
}
```

D.20 UserAppCreateFile Java Code

```
public class UserAppCreateFile implements Runnable {

 private CreateFile createFile = null;
 private DOStore doStore = null;
 private OpenFileStore openFileStore = null;
 private ErrorLog errorLog = null;

 public UserAppCreateFile(DOStore doStor, OpenFileStore openFileStor,
    ErrorLog errorLg) {
  doStore = doStor; openFileStore = openFileStor; errorLog = errorLg;
 }

 public void run() {
  createFile = new CreateFile(doStore, openFileStore, 0, 1, 0, errorLog);
  new Thread(createFile).start(); /* ua1 */
 }
}
```

D.21 UserAppWriteFile Java Code

```
public class UserAppWriteFile implements Runnable {

 private OpenFileStore openFileStore = null;
 private UserBuffer buff1 = null;
 private WriteFile writeFile1 = null;
 private int data;
 private ErrorLog errorLog = null;
```

```
 public UserAppWriteFile(OpenFileStore openFileStor, ErrorLog errorLg) {
  openFileStore = openFileStor; data = 65; errorLog = errorLg;
 }

 public void run() {
  buff1 = new UserBuffer(); /* uaw1 */
  while (data < 70) {
   buff1.add(data); /* uaw2 */
   data = data + 1; /* uaw3 */
  }
  writeFile1 = new WriteFile(openFileStore, 0, buff1, 5, errorLog);
  new Thread(writeFile1).start(); /* uaw4 */
 }
}
```

D.22 UserAppReadFile Java Code

```
public class UserAppReadFile implements Runnable {

 private OpenFileStore openFileStore = null;
 private UserBuffer buff1 = null;
 private ReadFile readFile1 = null;
 private ErrorLog errorLog = null;

 public UserAppReadFile(OpenFileStore openFileStor, ErrorLog errorLg) {
  openFileStore = openFileStor; errorLog = errorLg;
 }

 public void run() {
  buff1 = new UserBuffer(); /* uar1 */
  readFile1 = new ReadFile(openFileStore, 0, buff1, 5, errorLog);
  new Thread(readFile1).start(); /* uar2 */
 }
}
```

D.23 FileDirInfo Java Code

```
public class FileDirInfo {
```

```
 private int fileOffset;
 private int id;

 public FileDirInfo(int nameID) {
  fileOffset = 0; id = nameID;
 }

 public synchronized int getID() {
  return id;
 }
}
```

D.24 DataObject Java Code

```
public class DataObject {

 private int type;
 private FileDirInfo fileDirInfo = null;
 private int[] data = new int[10];
 private int freeSpace;

 public DataObject(int typ, FileDirInfo fileDirInf) {
  type = typ; fileDirInfo = fileDirInf; freeSpace = 10;
 }

 public synchronized FileDirInfo getFileDirInfo() {
  try {
   while (!(fileDirInfo != null)) {
    wait();
   }
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  return fileDirInfo;
 }

 public synchronized int read(int offset) {
  try {
   while (!(offset >= 0 && offset < 10)) {
```

```
      wait();
     }
    } catch (InterruptedException e) {
     e.printStackTrace();
    }
    return data[offset];
  }

  public synchronized void write(int val, int offset) {
    try {
     while (!(offset >= 0 && offset < 10)) {
      wait();
     }
    } catch (InterruptedException e) {
     e.printStackTrace();
    }
    data[offset] = val;
    notifyAll();
  }

  public synchronized int getType() {
    return type;
  }

  public synchronized int reserveSpace() {
    int initial_freeSpace = freeSpace;
    freeSpace = initial_freeSpace - 1;
    notifyAll();
    return initial_freeSpace;
  }

  public synchronized void unReserve() {
    freeSpace = freeSpace + 1;
    notifyAll();
  }
}
```

D.25 OpenFileInfo Java Code

```
public class OpenFileInfo {
```

```
private int accessMode;
private int shareMode;
private int fileOffset;
private DataObject dataObject = null;

public OpenFileInfo(int aMode, int sMode, DataObject dataObj) {
 shareMode = sMode; accessMode = aMode; fileOffset = 0;
 dataObject = dataObj;
}

public synchronized int getOffset() {
 return fileOffset;
}

public synchronized DataObject getDataObject() {
 try {
  while (!(dataObject != null)) {
   wait();
  }
 } catch (InterruptedException e) {
  e.printStackTrace();
 }
 return dataObject;
}

public synchronized void resetOffset() {
 fileOffset = 0;
 notifyAll();
}

public synchronized void incOffset() {
 fileOffset = fileOffset + 1;
 notifyAll();
}

public synchronized int getAccessMode() {
 return accessMode;
}

public synchronized int getShareMode() {
```

```
  return shareMode;
 }
}
```

D.26 DOStore Java Code

```
public class DOStore {

 private DataObject[] doArray = new DataObject[5];
 private int size;
 private int capacity;
 private int freeSpace;

 public DOStore() {
  size = 0; capacity = 5; freeSpace = 5;
 }

 public synchronized void add(DataObject f) {
  try {
   while (!(size >= 0 && size < capacity && capacity == 5)) {
    wait();
   }
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  doArray[size] = f;
  size = size + 1;
  notifyAll();
 }

 public synchronized DataObject getAtIndex(int indx) {
  try {
   while (!(indx >= 0 && indx < size && doArray[indx] != null)) {
    wait();
   }
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  return doArray[indx];
 }
```

```
 public synchronized int reserveSpace() {
  int initial_freeSpace = freeSpace;
  freeSpace = initial_freeSpace - 1;
  notifyAll();
  return initial_freeSpace;
 }

 public synchronized void unReserve() {
  freeSpace = freeSpace + 1;
  notifyAll();
 }

 public synchronized int getSize() {
  return size;
 }
}
```

D.27 OpenFileStore Java Code

```
public class OpenFileStore {

 private OpenFileInfo[] openArray = new OpenFileInfo[5];
 private int size;
 private int capacity;
 private int freeSpace;

 public OpenFileStore() {
  size = 0; capacity = 5; freeSpace = 5;
 }

 public synchronized void add(OpenFileInfo f) {
  try {
   while (!(size >= 0 && size < capacity && capacity == 5)) {
    wait();
   }
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  openArray[size] = f;
```

```
  size = size + 1;
  notifyAll();
 }

 public synchronized OpenFileInfo getAtIndex(int indx) {
  try {
   while (!(indx >= 0 && indx < size && openArray[indx] != null)) {
    wait();
   }
  }catch (InterruptedException e) {
   e.printStackTrace();
  }
  return openArray[indx];
 }

 public synchronized int reserveSpace() {
  int initial_freeSpace = freeSpace;
  freeSpace = initial_freeSpace - 1;
  notifyAll();
  return initial_freeSpace;
 }

 public synchronized void unReserve() {
  freeSpace = freeSpace + 1;
  notifyAll();
 }

 public synchronized int getSize() {
  return size;
 }
}
```

D.28 UserBuffer Java Code

```
public class UserBuffer {

 private int[] buffer = new int[10];
 private int capacity;
 private int size;
```

```
 public UserBuffer() {
  capacity = 10; size = 0;
 }

 public synchronized void add(int val) {
  try {
   while (!(size >= 0 && size < capacity && capacity == 10)) {
    wait();
   }
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  buffer[size] = val;
  size = size + 1;
  notifyAll();
 }

 public synchronized int get(int indx) {
  try {
   while (!(indx >= 0 && indx < capacity && capacity == 10)) {
    wait();
   }
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
   return buffer[indx];
 }
}
```

D.29 ErrorLog Java Code

```
public class ErrorLog {

 private int[] error = new int[5];
 private int size;
 private int lastIndex;

 public ErrorLog() {
  size = 0; lastIndex = -1;
 }
```

```
 public synchronized void add(int errorCode) {
  int initial_size = size;
  try {
   while (!(size >= 0 && size < 5)) {
    wait();
    initial_size = size;
   }
  } catch (InterruptedException e) {
     e.printStackTrace();
  }
  error[initial_size] = errorCode;
  size = initial_size + 1;
  lastIndex = initial_size;
  notifyAll();
 }

 public synchronized int getLast() {
  return error[lastIndex];
 }

 public synchronized void removeLast() {
  try {
   while (!(size > 0 && size <= 5)) {
    wait();
   }
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  error[lastIndex] = 0;
  size = size - 1;
  lastIndex = lastIndex - 1;
  notifyAll();
 }
}
```

Appendix E

Tooling

E.1 Pop up Menu - Translate Implementation

```
public class TranslatorAction implements IObjectActionDelegate {
  IStructuredSelection selection;
 ....
 public void run(IAction action) {
  if(selection.getFirstElement() instanceof MainClassImpl){
    MainClass main = (MainClass)selection.getFirstElement();
    Display display = Display.getCurrent();
    Shell shell = new Shell(display);
    Diagnostic diagnostic = Diagnostician.INSTANCE.validate(main);
    if(diagnostic.getSeverity()!=Diagnostic.OK){
      for (Iterator<Diagnostic> i=diagnostic.getChildren().iterator();
          i.hasNext();) {
        Diagnostic childDiagnostic = (Diagnostic)i.next();
        String msg=childDiagnostic.getMessage().replaceAll("of [^ ]*", "");
        MessageDialog.openInformation(
        shell,
        "OCB Plug-in",
        "Error in: "+ childDiagnostic.getData().get(0)+" "+ msg);
      }
    }else{
      EventBManager e = new EventBManager(main, true);
      e.translate();
      JavaManager j = new JavaManager(main);
      j.translate();
    }}}...}
```

E.2 OCB to Event-B Translation: OCBSequence

```
// Rule 3.7 process an OCB sequence
public void processOCBSequence(OCBSequence sequence, String endLabel)
    throws Exception {
  processNonAtomicClause(sequence.getLeftBranch(),
     sequence.getRightBranch().startLabel());
  processNonAtomicClause(sequence.getRightBranch(), endLabel);
}
```

Bibliography

[1] Multiverse - A Java based STM Implementation. Available at http://code.google.com/p/multiverse/.

[2] SPARKAda. Available at http://www.praxis-his.com/sparkada/index.asp.

[3] The HOL Website. available at http://hol.sourceforge.net/.

[4] XSTM - Software Transactional Memory. Available at http://www.xstm.net/stm.html.

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

[6] J.R. Abrial. Event Driven System Construction. Internal document, available at http://www.atelierb.eu/php/documents-en.php, 1999.

[7] J.R. Abrial. Event Driven Sequential Program Construction. Internal document, available at http://www.atelierb.eu/php/documents-en.php, 2001.

[8] J.R. Abrial. Discrete System Models. Internal document, available at http://www.atelierb.eu/php/documents-en.php, 2004.

[9] J.R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An Open Extensible Tool Environment for Event-B. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006.

[10] J.R. Abrial and D. Cansell. Click'n Prove: Interactive Proofs within Set Theory. In *TPHOLs*, 2003.

[11] J.R. Abrial, S.A. Schuman, and B. Meyer. Specification Language. In *On the Construction of Programs*, pages 343–410. 1980.

[12] A.Edmunds and M.Butler. Tool Support for Event-B Code Generation (to be presented at WS-TBFM2010).

[13] G. R. Andrews, R. A. Olsson, M. H. Coffin, I. Elshoff, K. D. Nilsen, T. D. M. Purdin, and G. M. Townsend. An Overview of the SR Language and Implementation. *ACM Trans. Program. Lang. Syst.*, 10(1):51–86, 1988.

[14] C. Aniszczyk. Using GEF with EMF. Available at http://www.eclipse.org/articles/Article-GEF-EMF/gef-emf.html.

[15] M. Anlauff. XASM - An Extensible, Component-Based ASM Language. In Gurevich et al. [70], pages 69–90.

[16] C. Artho, K. Havelund, and A. Biere. Using Block-Local Atomicity to Detect Stale-Value Concurrency Errors. In Farn Wang, editor, *ATVA*, volume 3299 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2004.

[17] B-Core(UK)Ltd. The B-Toolkit. Available at http://www.b-core.com.

[18] R.J. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, Abo Akademi, Department of Computer Science, Helsinki, Finland, 1978. Report A–1978–4.

[19] R.J. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, The Netherlands, 1980.

[20] R.J. Back, A. Mikhajlova, and J. von Wright. Class Refinement as Semantics of Correct Object Substitutability. *Formal Aspects of Computing*, 12(1), 2000.

[21] R.J. Back and J. von Wright. Contracts, Games and Refinement. *Electr. Notes Theor. Comput. Sci.*, 7, 1997.

[22] D.F. Bacon, R.E. Strom, and A. Tarafdar. Guava: A Dialect of Java Without Data Races. In *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 382–400, 2000.

[23] M. Barnett, K.R.M Leino, and W. Schulte. *The Spec# Programming System: An Overview*, volume 3362/2005 of *Lecture Notes in Computer Science*, pages 49–69. Springer, Berlin / Heidelberg, January 2005.

[24] B. Beckert, R. Hähnle, and P.H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[25] C. Boyapati, R. Lee, and M.C. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, pages 211–230, 2002.

[26] Cees-Bart Breunesse, Bart Jacobs, and Joachim van den Berg. Specifying and Verifying a Decimal Representation in Java for Smart Cards. In Hélène Kirchner and Christophe Ringeissen, editors, *AMAST*, volume 2422 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2002.

[27] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T.J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[28] L. Burdy, Y. Cheon, D.R. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, 7(3), 2005.

[29] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An Overview of JML Tools and Applications. *STTT*, 7(3):212–232, 2005.

[30] M. Butler. csp2B: A Practical Approach To Combining CSP and B. *Formal Aspects of Computing*, 12(3), 2000.

[31] M. Butler. A System-Based Approach to the Formal Development of Embedded Controllers for a Railway. *Design Automation For Embedded Systems*, 6, 2002.

[32] M. Butler. Incremental Design of Distributed Systems with Event-B, November 2008.

[33] M. Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.

[34] M. Butler and D. Yadav. An Incremental Development of the Mondex System in Event-B. *Formal Aspects of Computing*, 20(1):61–77, January 2008.

[35] M.J. Butler and M. Leuschel. Combining CSP and B for Specification and Property Verification. In Fitzgerald et al. [60], pages 221–236.

[36] A. Butterfield and J. Woodcock. Formalising Flash Memory: First Steps. In *ICECCS*, pages 251–260. IEEE Computer Society, 2007.

[37] M. Campione, K. Walrath, P. Chan, R. Lee, J. Kanerva, J. Gosling, B. Joy, G. Steele, G. Bracha, Technical Advisors - K. Arnold, T. Lindholm, and F. Yellin. The Java Language Specification - Second Edition, 2000.

[38] D. Cansell. The Click_n_Prove interface. from http://www.loria.fr/ cansell/cnp.html.

[39] G. Del Castillo and K. Winter. Model Checking Support for the ASM High-level Language. *Proc. of 6th Int. Conference for Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 331–346, 2000.

[40] O. Celiku and J. von Wright. Correctness and Refinement of Dually Nondeterministic Programs. Technical Report 516, TUCS, Mar 2003.

[41] ClearSy. B4Free. from http://www.b4free.com.

[42] ClearSy System Engineering. *Atelier B Translators*, version 4.6 edition.

[43] ClearSy System Engineering. *The B Language Reference Manual*, version 4.6 edition.

[44] ClearSy System Engineering. *The B Language Reference Manual*, version 1.8.5 edition.

[45] K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An Automatic Verification Tool for UML. Technical Report CSE-TR-423-00, CSE-TR-423-00, 2000.

[46] CSK Systems Corporation. The VDM++ Language Manual.

[47] D. Crocker. Safe Object-Oriented Software: The Verified Design-By-Contract Paradigm. In *Practical Elements of Safety: Proceedings of the Twelfth Safety-Critical Systems Symposium*, 2004.

[48] K. Damchoom, M. Butler, and J.R. Abrial. Modelling and Proof of a Tree-structured File System. In *ICFEM 2008*, volume LNCS 5256, pages 25–44. Springer, October 2008. Springer LNCS 5256.

[49] J. Davies, C. Crichton, E. Crichton, D. Neilson, and I.H. Sørensen. Formality, Evolution, and Model-driven Software Engineering. *Electr. Notes Theor. Comput. Sci.*, 130:39–55, 2005.

[50] E.W. Dijkstra. Chapter I: Notes on Structured Programming. pages 1–82, 1972.

[51] E.W. Dijkstra. Guarded Commands, Non-determinacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.

[52] E.W. Dijkstra. Guarded Commands, Non-determinancy and a Calculus for the Derivation of Programs. In F.L. Bauer and K. Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 1975.

[53] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[54] A. Edmunds and M. Butler. Linking Event-B and Concurrent Object-Oriented Programs. In *Refine 2008 - International Refinement Workshop*, May 2008.

[55] Escher Technologies. PerfectDeveloper. Available at http://www.eschertech.com.

[56] W. Reif et al. Structured Specifications and Interactive Proofs with KIV. In Wolfgang Bibel and Peter H. Schmidt, editors, *Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques*. Kluwer Academic Publishers, Dordrecht, 1998.

[57] FAA/NASA. OOTiA - Object Orientated Technology in Aviation Program. Available at http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot/.

[58] M.A. Ferreira, S.S. Silva, and J.N. Oliveira. Verifying Intel Flash File System Core Specification. *The Fourth Overture/VDM++ Workshop at FM2008*, 2008.

[59] J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag Telos, 2005.

[60] John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors. *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*. Springer, 2005.

[61] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, pages 234–245, 2002.

[62] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. *ACM SIGPLAN Notices*, 38(5):338–349, 2003.

[63] Formal Systems Europe Ltd. FDR Model Checker. Available from http://www.fsel.com.

[64] D. Frankel. *Model Driven Architecture : Applying MDA to Enterprise Computing*. Wiley, 2003.

[65] A. Freitas and A. Cavalcanti. Automatic Translation from Circus to Java. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2006.

[66] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the Verification Grand Challenge: A Roadmap. *iceccs*, 0:153–162, 2008.

[67] J. Gosling, B. Joy, and G. Steele. *The Java (TM) Language Specification*. Addison-Wesley, 1996.

[68] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification - Third Edition*. Addison-Wesley, 2004.

[69] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–37. Oxford University Press, 1994.

[70] Y. Gurevich, P.W. Kutter, M. Odersky, and L. Thiele, editors. *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings*, volume 1912 of *Lecture Notes in Computer Science*. Springer, 2000.

[71] Y. Gurevich, W. Schulte, and C. Wallace. Investigating Java Concurrency Using Abstract State Machines. In Gurevich et al. [70], pages 151–176.

[72] S. Hallerstede. Justifications for the Event-B Modelling Notation. In Julliand and Kouchnarenko [91], pages 49–63.

[73] T. Harris. Exceptions and Side-effects in Atomic Blocks. *Sci. Comput. Program.*, 58(3):325–343, 2005.

[74] T. Harris and K. Fraser. Language Support for Lightweight Transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

[75] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[76] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969.

[77] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Commun. ACM*, 17(10):549–557, 1974.

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

[79] G.J. Holzmann. The Model Checker SPIN. *Software Engineering, IEEE Transactions*, 23(5), 1997.

[80] D. Hovemeyer and W. Pugh. Finding Concurrency Bugs in Java. In *In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[81] D. Hovemeyer, W. Pugh, and J. Spacco. Atomic Instructions in Java. In *In Magnusson [14*, pages 133–154.

[82] Intel Corporation. Intel Flash File System Core Reference Guide. Available at http://sunsite.rediris.es/pub/mirror/intel/flcomp/manuals/30443601.pdf.

[83] Intel Corporation et al. Open NAND Flash Interface Specification. Available at http://www.onfi.org/.

[84] D. Jackson. An Intermedicate Design Language and Its Analysis. In *SIGSOFT FSE*, pages 121–130, 1998.

[85] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[86] M. A Jackson. *System development (Prentice-Hall International series in computer science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1983.

[87] B. Jacobs, K.R.M. Leino, and W. Schulte. Verification of Multithreaded Object-oriented Programs with Invariants. In *Proceedings of the workshop on Specification and Verification of Component-Based Systems - SAVCBS2004*, 2004.

[88] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs. In *Eighth International Conference on Formal Engineering Methods (ICFEM 2006)*, 2006.

[89] I. Johnson, C. Snook, A. Edmunds, and M. Butler. Rigorous Development of Reusable, Domain-specific Components, for Complex Applications. *Proceedings of 3rd International Workshop on Critical Systems Development with UML*, pages 115–129, 2004.

[90] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[91] J. Julliand and O. Kouchnarenko, editors. *B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2006.

[92] A.W. Keen, T. Ge, J.T. Maris, and R.A. Olsson. JR: Flexible Distributed Programming in an Extended Java. *ACM Trans. Program. Lang. Syst.*, 26(3):578–608, 2004.

[93] B.W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition.* Prentice-Hall, 1988.

[94] J. Kienzle. On Atomicity and Software Development. *Journal of Universal Computer Science*, 11(5), 2005.

[95] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[96] L. Lamport. The $^{+}$CAL Algorithm Language. In E. Najm, J.F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, page 23. Springer, 2006.

[97] L. Lamport. Checking a Multithreaded Algorithm with $^{+}$CAL. In S. Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2006.

[98] K. Lano, K. Androutsopoulos, and D. Clark. Structuring and Design of Reactive Systems Using RSDS and B. In T.S.E. Maibaum, editor, *FASE*, volume 1783 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2000.

[99] K. Lano, D. Clark, and K. Androutsopoulos. UML to B: Formal Verification of Object-Oriented Models. In *IFM*, pages 187–206, 2004.

[100] D. Lea. *Concurrent Programming in Java (Second Edition): Design Principles and Patterns.* Addison-Wesley, 2004.

[101] D. Lea. The java.util.concurrent Synchronizer Framework. *Science of Computer Programming*, 58(3), 2005.

[102] G.T. Leavens and Y. Cheon. Design by Contract with JML. Draft Paper from http://www.cs.iastate.edu/ leavens/JML.

[103] F. Lerda and W. Visser. Addressing Dynamic Issues of Program Model Checking. In Matthew B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 80–102. Springer, 2001.

[104] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *Proceedings of Formal Methods Europe 2003*, 2003.

[105] A.M. Lister. The Problem of Nested Monitor Calls. *Operating Systems Review*, 11(3):5–7, 1977.

[106] M. Scheidgen. The Textual Editing Framework. Available at http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/index.html.

[107] I. Maamria, M. Butler, A. Edmunds, and A. Rezazadeh. On an Extensible Rule-based Prover for Event-B. In *ABZ2010*, February 2010.

[108] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 1999.

[109] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers Norwell, MA, USA, 1993.

[110] J.P. Mermet, editor. *UML-B Specification for Proven Embedded Systems Design*. Kluwer, 2004.

[111] B. Meyer. *Eiffel : The Language*. Prentice-Hall, 1992.

[112] B. Meyer. The Start of an Eiffel Standard. *Journal of Object Technology*, 1(2):95–99, 2002.

[113] L. Mikhajlov. Software Reuse Mechanisms and Techniques: Safety Versus Flexibility. Turku Centre for Computer Science, TUCS Dissertations, December 1999.

[114] A. Mikhajlova. Ensuring Correctness of Object and Component Systems. Turku Centre for Computer Science, TUCS Dissertations, October 1999.

[115] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[116] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[117] C. Morgan. Procedures, Parameters, and Abstraction: Separate Concerns. *Sci. Comput. Program.*, 11(1):17–27, 1988.

[118] C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, June 1994.

[119] M.Williams. *Microsoft Visual C# .NET (Core Reference)* . Microsoft Press, 2002.

[120] Object Management Group (OMG). Object Constraint Language Specification. Available at http://www.omg.org/technology/documents/formal/ocl.htm.

[121] Object Management Group (OMG). UML 2.0 Superstructure specification. Available at http://www.omg.org/technology/uml/index.htm.

[122] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752, London, UK, 1992. Springer-Verlag.

[123] P.H.Welch, J.R. Aldous, and J. Foster. CSP Networking for Java (JCSP.net). In *Computational Science - ICCS 2002: International Conference*, 2002.

[124] Project MATISSE: Methodologies and Technologies for Industrial Strength Systems Engineering. *Event B Reference Manual*. IST-1999-11435.

[125] R Razali, C. F. Snook, and M. R. Poppleton. Comprehensibility of UML-based Formal Model ? A Series of Controlled Experiments. In *1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech) 2007*, pages 25–30, November 2007.

[126] R. Razali, C. F. Snook, M. R. Poppleton, P. W. Garratt, and R. J. Walters. Experimental Comparison of the Comprehensibility of a UML-based Formal Specification versus a Textual One. In B. Kitchenham, P. Brereton, and M. Turner, editors, *11th International Conference on Evaluation and Assessment in Software Engineering (EASE'07)*, pages 1–11. British Computer Society (BCS), 2007.

[127] E. Rodríguez, M.B. Dwyer, C. Flanagan, J. Hatcliff, G.T. Leavens, and Robby. Extending JML for Modular Specification and Verification of Multi-threaded Programs. In A.P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 551–576. Springer, 2005.

[128] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[129] M. Saaltink. Z and Eves. In J. E. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 223–242. Springer, 1991.

[130] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*, pages 27–37, 1997.

[131] S. Schneider and H. Treharne. Communicating B Machines. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 416–435. Springer, 2002.

[132] S. Schneider and H. Treharne. Verifying Controlled Components. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 87–107. Springer, 2004.

[133] S. Schneider and H. Treharne. CSP Theorems for Communicating B Machines. *Formal Asp. Comput.*, 17(4):390–422, 2005.

[134] SGS-Thomson Microelectronics Ltd. *Occam 2.1 Reference Manual*, 1995.

[135] G. Smith. *The Object-Z Specification Language (Advances in Formal Methods)*. Springer, December 1999.

[136] G. Smith. *The Object-Z specification language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[137] C. Snook and M. Butler. Deliverable D4.1.3 : Final Tool Extensions for Integration of UML and B. from Project IST-2000-30103, PUSSEE - Paradigm Unifying System Specification Environments for Proven Electronic design.

[138] C. Snook and M. Butler. *U2B - A tool for translating UML-B models into B*, volume UML-B Specification for Proven Embedded Systems Design. Springer, 2004.

[139] C. Snook and M. Butler. UML-B: Formal modeling and Design Aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

[140] C. Snook and M. Butler. UML-B: Formal Modelling and Design Aided by UML. *ACM Transactions on Software Engineering and Methodology*, 2006.

[141] C. Snook and M. Butler. UML-B and Event-B: An Integration of Languages and Tools. In *The IASTED International Conference on Software Engineering - SE2008*, February 2008.

[142] C. Snook, M. Butler, and I. Oliver. Towards a UML Profile for UML-B. Technical report, Electronics and Computer Science, University of Southampton, 2003.

[143] J. M. Spivey. Understanding Z: A Specification Language and its Formal Semantics. *Cambridge Tracts in Theoretical Computer Science*, 3, 1988.

[144] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[145] B. Stevens. Implementing Object-Z with Perfect Developer. *Journal of Object Technology*, 5(2):189–202, 2006.

[146] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, third edition, February 2000.

[147] T.S. Taft, R.A. Tucker, R.L. Brukardt, and E. Ploedereder, editors. *Consolidated Ada reference manual: language and standard libraries*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[148] The Eclipse Project. Eclipse - an Open Development Platform. Available at http://www.eclipse.org/.

[149] The Eclipse Project Team. Eclipse Java Development Tools (JDT) Subproject. Available at http://www.eclipse.org/jdt/.

[150] The Eclipse Project Team. Eclipse Modeling Project. Available at http://www.eclipse.org/modeling/.

[151] The IEEE and The Open Group. The Open Group Base Specifications Issue 7 . IEEE Std 1003.1^{TM}-2008. Available at http://www.opengroup.org/onlinepubs/9699919799/.

[152] The RAISE Team. *The RAISE Method*. Prentice-Hall, 1999.

[153] The RODIN Project. Available at http://rodin.cs.ncl.ac.uk.

[154] E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[155] G. van Rossum and F.L. Drake Jr. *The Python Language Reference Manual (version 2.5)*. Network theory Ltd., 2006.

[156] W. Visser, K. Havelund, G.P. Brat, S. Park, and F. Lerda. Model Checking Programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[157] P.H. Welch and J.M.R. Martin. A CSP Model for Java Multithreading. In *Software Engineering for Parallel and Distributed Systems*, 2000.

[158] P.H. Welch and J.M.R. Martin. Formal Analysis of Concurrent Java Systems. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 275–301, sep 2000.

[159] J. Woodcock and A. Cavalcanti. A Concurrent Language for Refinement. In A. Butterfield, G. Strong, and C. Pahl, editors, *IWFM*, Workshops in Computing. BCS, 2001.

[160] Jim Woodcock, Ana Cavalcanti, and Leonardo Freitas. Operational Semantics for Model Checking Circus. In Fitzgerald et al. [60], pages 237–252.

[161] D. Yadav and M. Butler. Formal Development of Fault Tolerant Transactions for a Replicated Database using Ordered Broadcasts. In *Methods, Models and Tools for Fault Tolerance (MeMoT 2007)*, pages 33–42, 2007.

[162] L. Yang and M. Poppleton. Automatic Translation from Combined B and CSP Specification to Java Programs. In Julliand and Kouchnarenko [91], pages 64–78.