Linking Event-B and Concurrent Object-Oriented Programs

Andrew Edmunds[1]

School of Electronics and Computer Science,
University of Southampton, Highfield, Southampton, SO17 1BJ, UK

Michael Butler[2]

School of Electronics and Computer Science,
University of Southampton, Highfield, Southampton, SO17 1BJ, UK

Abstract

The Event-B method is a formal approach to modelling systems, using refinement. Initial specification is done at a high level of abstraction; detail is added in refinement steps as the development proceeds toward implementation. In software systems that use concurrent processing it is necessary to provide details of concurrent features before implementation. Our contribution is to show how Event-B models can be linked to concurrent, object-oriented implementations using an intermediate, object-oriented style specification notation. To validate our approach and gain further insight we automated the translation process with an Eclipse plug-in which produces an Event-B model and Java code. We call the new notation Object-oriented Concurrent-B (OC-B). The notation facilitates specification of the concurrent aspects of a development, and facilitates reasoning about concurrency issues in an abstract manner. We abstract away implementation details, such as locking, and provide the developer with a clear view of atomicity using labelled atomic clauses. We build on techniques introduced in UML-B to model object-oriented developments, introducing non-atomic operations and features for specifying implementation level details.

Keywords: Event-B, Object-oriented, Concurrency, Refinement

1 Introduction

The Event-B method [2] is a formal approach to modelling systems, with tool support [3]. The modelling approach uses an event based view of how a system evolves atomically, from one state to another. The Event-B approach has evolved from classical-B [1] which was targeted more specifically at modelling software systems. In Event-B a system's state is modelled using sets, constants and variables; and updates to state are described in the bodies of guarded events. System properties are specified in invariants and proof obligations are generated, which should be discharged in order to prove that the event actions do not violate the invariants. The

[1] Email: ae03r@ecs.soton.ac.uk

[2] Email: mjb@ecs.soton.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Event-B approach uses refinement to link an abstract model with successively more concrete models, and a linking invariant relates the state of the concrete model with its abstract counterpart.

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. Consideration is given to how tasks may be performed by executing processes, and how the processes may interleave. We shall use Java [8] 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. One of our contributions is the introduction of an intermediate specification language, Object-oriented, Concurrent-B (OC-B), which we use to link Event-B models and object-oriented implementations. The new notation sits at the interface between the two technologies, and we incorporate aspects of both. Another contribution is a translation tool which we developed to perform translation of the intermediate notation to an Event-B model and to Java code. We would expect to show that the Event-B model refines an existing model in order to show that it satisfies properties of the abstraction. 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 [12], which we map to program counters.

When defining the mapping to Java we need to ensure freedom from interference by restricting visibility of its data, and enforce a mutual exclusion policy for access to shared data. It also utilises conditional waiting, but incorporates restrictions to avoid the nested monitor problem [15] (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; 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 [13].

1.1 An Overview of OC-B

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. An event, at a higher level of abstraction can contain a number of updates to the state, which occur atomically. Using OC-B notation we can specify a non-atomic operation where updates occur in a number of interleaving atomic steps. Each of these atomic steps maps to an event in the abstract model. To facilitate the interleaving behaviour we introduce a sequential operator,';', and the notion of non-atomic operations. Processes may run concurrently and may interleave at the point of a sequential operator, and at defined points within looping and branching clauses. 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, 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 non-recursive, atomic procedure calls. We also add the restriction that monitors are not able to refer to processes or other monitors, which will prevent the nested monitor problem.

Section 2 introduces the definition of a system with processes and program counters. Section 3 introduces our definition of monitors, which we extend in section 4 with object-oriented features. Section 5 introduces some syntactic sugar, and we show an example refinement of an Event-B model to an OC-B model in section 6. Section 7 describes the mapping to Java, section 8 discusses related work and we present conclusions and ideas for future work in section 9.

2 Mapping Processes to Event-B

We begin by introducing the notion of processes and program counters using a syntax based on the guarded command language [6]. A system may have a number of processes defined, each with a non-atomic operation that is able to interleave with non-atomic operations of other processes.

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

The syntax of a non-atomic clause allows a sequence, choice, loop or atomic statement. Atomic statements have a guarded body as follows, where the body consists of (for the moment) *Action* clauses involving assignments. Assignments are of the form $x := E$, where x is a variable name and E is an expression, and may be composed using, $\|$, the parallel operator.

$$Atomic ::= \quad StartLabel :< [Guard \rightarrow] Body >$$

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, $l1 :< y := x >\ ;\ l2 :< x := x + 1 >$, which results in the following two events,

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

This event 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.

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

This event is enabled when the program counter value is *l2*. Once again state updates are contained in the event body. Since no clauses follow *l2* in the sequence, the next program counter value is dependent upon the clause that contains it, if one exists. If the sequence clause is not contained in another clause then the process terminates. In this case the value supplied as a parameter during translation indicates process termination; we arbitrarily choose a constant label, *terminated*, each process will have such a terminating label.

The processes of a system are defined as a function over process names to processes.

Definition 2.1 $Processes = (PName \rightarrow Process)$

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

Definition 2.2 $Process = \mathbb{P}(PVar) \times NonAtomic$

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 2.3 $TP \in Process \times PName \rightarrow \mathbb{P}(Event)$

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 2.4 $TNA \in NonAtomic \times Label \times PName \rightarrow \mathbb{P}(Events)$

A process with variables *var*, body *na* and name P is mapped to a set of events by applying *TNA* to the body. *tp* is a constant label indicating a termination state for a process.

Definition 2.5 $TP((var, na), P) = TNA(na, tp, P)$

We now look at the non-atomic syntactic elements; firstly the sequence clause. *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. The definitions involving the non-atomic clauses are well-defined if the start label of each operand differs; except for the choice construct, where each label must be the same. We assume that the function, $sLabel \in NonAtomic \rightarrow Label$, yields the first label of a non-atomic clause.

Definition 2.6
$$TNA(na1; na2, l2, P) = TNA(na1, l1, P) \cup TNA(na2, l2, P)$$
$$\text{where } l1 = sLabel(na2)$$

A branching clause is defined as follows,

Definition 2.7 $TNA(na1[]na2, l2, P)$
$= TNA(na1, l2, P) \cup TNA(na2, l2, P)$

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 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 and the start label of the second clause are the same (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 loop body 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 2.8 $TNA(do\ l1 :< g \rightarrow b > od, l2, P)$
$= \{TLA(l1 :< g \rightarrow b >, l1, P)\}$
$\cup \{TLA(l1 :< \neg g \rightarrow Skip >, l2, P)\}$

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 2.9 $TNA(do\ l1 :< g \rightarrow b >; na\ od, l2, P)$
$= \{TLA(l1 :< g \rightarrow b >, l3, P)\}$
$\cup \{TLA(l1 :< \neg g \rightarrow Skip >, l2, P)\} \cup TNA(na, l1, P)$
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 an event. If the guard is omitted from the specification then a true guard is assumed.

Definition 2.10 $TLA \in Atomic \times Label \times PName \rightarrow Event$

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. We define the transformation of an atomic clause with a body consisting of actions A, as follows.

Definition 2.11 $TLA(l1 :< g \rightarrow A >, l2, P)$
$=$
WHEN $P_{\text{pc}} = l1 \wedge g$
THEN $A \parallel P_{\text{pc}} := l2$
END

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

3 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, non-recursive, atomic procedure calls, and atomic assignments. Procedures can have formal parameters, which we define as a sequence, *LVar*, of local variable declarations; these correspond with a sequence of actual parameters in the 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 [17,18]; 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. Therefore we need a way to identify both the monitor and the procedure in a call; we use dot notation to do this. This is not the same dot notation that we use later for object-oriented features; since here we are identifying a monitor name and not an instance. To be well defined the monitor must contain a procedure with the called name; and the actual parameters of the call, $a_1, \ldots, a_k$, must match the formal parameters, $f_1, \ldots, f_k$, of the procedure, in number and type. 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. A single **return** variable can be used on the left hand side (LHS) of an assignment statement in the procedure body, and 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.

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

Monitors is a collection of monitors over monitor names,

Definition 3.1 $Monitors = (MName \rightarrow Monitor)$

A monitor has a set of variables and some procedures,

Definition 3.2 $Monitor = \mathbb{P}(MVar) \times Procedures$

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

Definition 3.3 $Procedures = (PdName \rightarrow Procedure)$

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

Definition 3.4 $Procedure = LVar \times Guard \times Action \times T$

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 3.5 $TLA(l1 :< g_c \rightarrow v := m.pn(a_1, \ldots, a_k) >, l2, P)$
$=$
WHEN $P_{pc} = l1 \wedge g_p[f_1, \ldots, f_k \backslash a_1, \ldots, a_k] \wedge g_c$
THEN $A[f_1, \ldots, f_k \backslash a_1, \ldots, a_k][\textbf{return} \backslash v] \parallel P_{pc} := l2$
END

where procedure pn of monitor m is defined by $m.pn(f_1, \ldots, f_k) = g_p \rightarrow A$

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.

4 Mapping Object-Oriented Features to Event-B

We have previously introduced processes and monitors and until now there has only been one process or monitor associated with a given 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**. Each monitor and process class must have a constructor procedure where initialisation of variables takes place. A new instance is constructed when the **create** procedure is invoked. Actual parameters, *a*, supplied to constructors may be used to initialise variables, by substitution of formal parameters, *f*. A system is modelled as a class with the name *Main* and type *MainClass*, its non-atomic clause corresponds to the Java *main* method - the entry point for execution in the implementation.

Our approach uses techniques introduced in UML-B [23], 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 $C_{inst} \subseteq C$ to represent the current set of instances of C. Each variable declaration $v \in T$ of class C maps to a variable with the same name in event-B, and typed as $v \in C_{inst} \rightarrow T$. Our approach adds a program counter variable P_{pc} for class each process class P,

typed as $P_{\text{pc}} \in P_{\text{inst}} \rightarrow \mathit{Label}$.

Non-atomic operations contain labelled atomic clauses which map to events. Program counters values (derived from labels of the atomic clauses) are used to model the flow of execution through the non-atomic operations. It is assumed that process instantiations begin processing immediately, that is, in the implementation the threads are started immediately following creation. We add to the syntax an atomic constructor-call clause involving class C, and add *ProcessClass* and *MonitorClass*.

$$\begin{aligned} \mathit{Body} ::= \quad & \ldots \\ & \mid v := C.\mathbf{create}(a_1, \ldots, a_k) \end{aligned}$$

$$\mathit{ProcessClass} ::= \quad \mathit{PName}\ \mathit{PVar}^+\ \mathit{NonAtomic}\ \mathit{Constructor}$$

$$\mathit{MonitorClass} ::= \quad \mathit{MName}\ \mathit{MVar}^*\ \mathit{Procedure}^+\ \mathit{Constructor}$$

The *TNA* mapping function is modified to allow the additional *Main* class name but otherwise the type definition remains the same.

Definition 4.1 $TNA \in \mathit{NonAtomic} \times \mathit{Label} \times (\mathit{PName} \cup \{\mathit{Main}\}) \rightarrow \mathbb{P}(\mathit{Events})$

The *TLA* mapping function is modified to accommodate the object-oriented features,

Definition 4.2 $TLA \in \mathit{Atomic} \times \mathit{Label} \times (\mathit{PName} \cup \{\mathit{Main}\}) \rightarrow \mathit{Event}$

When a variable is used in an OC-B clause its use is with respect to the class in which it is used. When we map to the Event-B model we need to model the variable, and refer to it with respect to an instance. The instance may be the caller; the target, in the case of a procedure call; or a new instance, in the case of constructor initialisations. This occurs in both actions and guards of the mapping. If v is a variable, and s is an instance, of class C, then $v(s)$ refers to the value of variable v belonging to instance s. To rename a variable we apply the function, TV, which takes a guard, action or expression parameter, and maps it to the corresponding Event-B representation. We additionally supply a set of variable names (those of the class being referred to) and the name of the Event-B variable representing the instance. The type of TV is defined as follows,

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

We show an example mapping with a variable v, used in a labelled assignment $l1 : v := v + 1$, and a calling instance s. The mapping using TV is, $TV(v := v+1, vn, s) = (v(s) := v(s)+1)$ where $vn = \{v\}$. The effect of function application is that wherever a variable in vn occurs it is referred to with respect to s.

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 local variables, G is a guarding predicate, and the body S contains some assignment

actions. The following notation is used for any class C, C_{pc}(s), is the program counter for instances of class C. C_{inst} is the set of current instances of class C. C_{set} is the set of potential instances of class C. The new definition of TLA for a labelled atomic clause follows where the clause is defined in class Q,

Definition 4.4 $TLA(l1 :< g \rightarrow A >, l2, Q)$
$=$
ANY s
WHERE $s \in Q_{\mathrm{inst}} \wedge Q_{\mathrm{pc}}(s) = l1 \wedge TV(g, vn, s)$
THEN $TV(A, vn, s) \parallel Q_{\mathrm{pc}}(s) := l2$
END

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

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

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

where:
The constructor procedure body A is defined as follows, $P.create(f_1, \ldots, f_k) = A$.
A' is the action A with formal parameters substituted by actual parameters that refer to the calling instance,
$A' = A[f_1, \ldots, f_k \backslash TV(a_1, vq, s), \ldots, TV(a_k, vq, s)]$.
na is the non-atomic clause of class P.
vq and vp are sets of variable names, of the caller, and new instance respectively.

A similar mapping exists for monitor class instantiation, but excludes setting of a program counter, with $P_{\mathrm{pc}}(new) := sLabel(na)$; monitors do not have program counters since they play a passive role in the system.

We now look at the definition of TLA for monitor procedure calls. We define a call of procedure named pn on target m, a variable belonging to instance s; $j = m(s)$ types an Event-B local variable 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 on the RHS of assignment expressions in A.

Definition 4.6 $TLA(l1 :< g_{\mathrm{c}} \rightarrow v_1 := m.pn(a_1, \ldots, a_n) >, l2, Q)$
$=$
ANY s, j
WHERE $s \in Q_{\mathrm{inst}} \wedge Q_{\mathrm{pc}}(s) = l1 \wedge j = m(s) \wedge TV(g_{\mathrm{p}}', vj, j)$
$\wedge TV(g_{\mathrm{c}}, vq, s)$

THEN $TV(A', vj, j) \parallel Q_{\mathrm{pc}}(s) := l2$
END

where:
$m.pn(a_1, \ldots, a_k) = g_{\mathrm{p}} \rightarrow A$.
$g_{\mathrm{p}}{}'$ is the procedure guard, actual parameter variables refer to the calling instance, and substitutions are applied,
$g_{\mathrm{p}}{}' = \mathrm{g_p}[f_1, \ldots, f_k \backslash TV(a_1, vq, s), \ldots, TV(a_k, vq, s)]$.
A' is the action clause, actual parameter variables refer to the calling instance, and substitutions are applied,
$A' = A[f_1, \ldots, f_k \backslash TV(a_1, vq, s), \ldots, TV(a_k, vq, s)][\mathbf{return} \backslash v_1]$.
vq and vj are sets of variable names, of the caller, and monitor instance respectively
.

5 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. We specify a branching construct guard g_i and action b_i pairs are evaluated atomically; the following *andthen* clause can contain some subsequent non-atomic clause which may occur after the action of some other process due to interleaving. We know however that the variables referred to in the guard will not be changed unexpectedly by some external process due to the restrictions on their visibility.

Definition 5.1

$l1 :$ **if**(g_1) **then** b_1 **andthen** na_1 **endif**
elseif(g_2) **then** b_2 **andthen** na_2 **endelseif**$\ldots$
else b_n **andthen** na_n **endelse**
$=$
$l1 :< g_1 \rightarrow b_1 >; na_1$
$[]l1 :< \neg g_1 \wedge g_2 \rightarrow b_2 >; na_2 \ldots$
$[]l1 :< \neg g_1 \wedge \neg g_2 \wedge \ldots \wedge \neg g_n \rightarrow b_n >; na_n$

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

Definition 5.2

$l1 :$ **while**(g) **do** b **andthen** na **endwhile**
$=$
$do\ l1 :< g \rightarrow b >; na\ od$

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

Definition 5.3

when$(g_{\mathrm{p}})\{A\}$
$=$
$< g_{\mathrm{p}} \rightarrow A >$

6 An Example Refinement from Event-B to OC-B

We now show how the translation functions may be applied to give rise to Event-B. Firstly we show some details of an example that we may want to refine, perhaps an existing UML-B development. The abstract development models a set of processes and shared objects, with an event which repeatedly assigns the value of a variable val to i and records the number of attempts. The assignment is however only made if the assigned value is greater than zero (we assume some other event can change the value). The variables are declared and typed as follows,

VARIABLES
$Prc, Shared, val, attempts, i$
INVARIANT
$Prc \in \mathbb{P}(Prc_{\text{set}})$
$Shared \in \mathbb{P}(Shared_{\text{set}})$
$val \in Shared \rightarrow \mathbb{Z}$
$attempts \in Shared \rightarrow \mathbb{Z}$
$i \in Prc \rightarrow \mathbb{Z}$

The updates are described by the following event,

$UpdateI =$
ANY $s,\ m$
WHERE $s \in Prc \wedge m \in Shared \wedge val(m) > 0$
THEN $attempts(m) := attempts(m) + 1 \parallel i(s) := val(m)$
END

There are also two constructor events which model instance creation, we show only the constructor for Prc instances, the constructor for the $Shared$ class is similar.

$newProcess =$
ANY new
WHERE $new \in Prc_{\text{set}} \setminus Prc$
THEN $i(new) := 0 \parallel Prc := Prc \cup \{new\}$
END

In order to provide an implementation for this Event-B specification we need to add some detail to the development. We can see that the $UpdateI$ event can be performed repeatedly so we can refine the abstract event with looping behaviour by using a looping construct in the $ProcessClass$ instances. The state, and the updates to it, can be shared between processes so we specify a $MonitorClass$ to encapsulate this. State updates of the monitor are described in procedure definitions, the processes make use of these by invoking procedure calls. Fig. 1 shows the OC-B specification of a $ProcessClass$, Prc. The run operation of a $ProcessClass$

is used to describe its behaviour during its execution and contains one or more labelled atomic constructs. Here we see the looping construct, in which we place the update to the shared monitor. We place an arbitrary limit on the number of iterations for which we introduce the variable *count* to the OCB specification. The purpose of this is to demonstrate the looping feature and the occurrence of the false branch of a loop, since the loop could actually run forever.

The *ProcessClass* encapsulates the integer i from the abstraction, along with *shared* which represents the shared *MonitorClass*, and *count* which keeps track of the loop iterations. Each of the variables is initialised in the *create* method. The *while* construct describes the iterating behaviour, and calls the *getVal* procedure of *shared* which contains the updates to the monitor. The *andthen* clause permits further actions to be specified, we use it to update the *counter* variable.

```
ProcessClass Prc{
 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
   }
}
```

Fig. 1. An Example OC-B ProcessClass Specification

The *MonitorClass Shared* shown in Fig.2 encapsulates the variables *val* and *attempts* of the abstraction. *attempts* records the number of times the value has been read from the monitor. We only want to return values greater than zero, so the process blocks unless $val > 0$. The blocking behaviour is specified using the *when* construct and we assume some other process is performing updates that may unblock a process. The value returned by the *getVal* procedure is assigned to the process variable i at the point of the procedure invocation.

In order to construct an implementation we need to provide an entry point for processing. We do this using the *MainClass* specification shown in Fig. 3. In the *MainClass* we declare variables for the processes and shared monitor. We use labelled atomic constructs in the *main* operation to describe the order of instantiation, and sharing of monitors among processes. In the *main* operation we call the create procedures, and pass the shared monitor *sh* to the process constructors as a parameter. In the following example we show the constructor event, *evt_m2* that

```
MonitorClass Shared{
 Integer val, Integer attempts

 Procedure create(){
   val := 0 || attempts := 0
 }

 Procedure getVal(){
   when(val > 0){
     attempts := attempts + 1 || return := val
   }
 }:Integer

 Procedure setVal(Integer v){ val := v }
}
```

Fig. 2. The Example OC-B MonitorClass Specification

```
MainClass Main{
 Prc pr1, Prc pr2, Shared sh , ...

 Operation main(){
   m1: sh := Shared.create();
   m2: pr1 := Prc.create(sh);
   m3: pr2 := Prc.create(sh) ...
 }
}
```

Fig. 3. The Example OC-B MainClass Specification

arises from the **create** call labelled by *m2* in the *main* operation of the *MainClass*. A new process instance is created and initialised, and the program counter is updated, since there are no following labelled clauses the calling process terminates. This event refines the event *newProcess* of the abstract model, the mapping gives rise to one event per constructor clause, here we show the event corresponding to label $m2$.

$evt\_m2 =$
REFINES *newProcess*
ANY new, s
WHERE $s \in Main \wedge Main_{\mathrm{pc}}(s) = m2 \wedge new \in Prc_{\mathrm{set}} \setminus Prc$
THEN $shared(new) := sh(s) \parallel i(new) := 0 \parallel count(new) := 0 \parallel$
$Main_{\mathrm{pc}}(s) := tp \parallel Prc := Prc \cup \{new\} \parallel pr1(s) := new \parallel$
$Prc_{\mathrm{pc}}(new) := l1$
END

We now consider the translation of the *run* operation of *ProcessClass* which gives

rise to three events. The events arise from the translation of the first clause *l1* with the true guard; *l1* with the false guard; and the second clause *l2* of the body. We now show the event *evt_l1_true*, the true branch that arises from the clause labelled *l1*. The refinement consists of two aspects, one of which deals with the newly introduced program control structure; the other refines the existing abstract events; the following event refines $UpdateI$ of the abstract model.

$evt\_l1\_true =$
REFINES *Update1*
ANY s, m
WHERE $s \in Prc \wedge Prc_{\mathrm{pc}}(s) = l1 \wedge count(s) < 100 \wedge m = shared(s)$
$\wedge\, val(m) > 0$
THEN $attempts(m) := attempts(m) + 1 \parallel i(s) := val(m) \parallel Prc_{\mathrm{pc}}(s) := l2$
END

then the event *evt_l1_false* that arises due to the false guard, which refines *skip* since it has no counterpart in the abstraction.

$evt\_l1\_false =$
REFINES *SKIP*
ANY s
WHERE $s \in Prc \wedge Prc_{\mathrm{pc}}(s) = l1 \wedge \neg(count(s) < 100)$
THEN $Prc_{\mathrm{pc}}(s) := tp$
END

then the third the event, *evt_l2*, arising from the second the clause labelled *l2*,

$evt\_l2 =$
REFINES *SKIP*
ANY s
WHERE $s \in Prc \wedge Prc_{\mathrm{pc}}(s) = l2$
THEN $count(s) := count(s) + 1 \parallel Prc_{\mathrm{pc}}(s) := l1$
END

7 Mapping to Java

The mapping of OC-B to Java is mostly self-evident since we incorporate object-oriented aspects into the notation, so we elide most of the translation rules. We present an overview of the strategy followed by an example showing the result of a translation. The OC-B *system* maps to a Java class, in particular the non-atomic clause is mapped to a *Main* class and used to populate a *main* method. Each *ProcessClass* maps to a Java class that implements the *java.lang.Runnable* interface; and the non-atomic, *na*, clause maps to the *run* method body. *MonitorClass* maps

to a Java class that does not implement the *Runnable* interface (since they are not required to behave as threads). Each monitor procedure maps to a synchronized Java method that can be called by the processes. Instance variables declared in *system*, and in class definitions, map to Java fields with private visibility. Access using Java synchronized methods ensures their use is free from interference.

The *when* clause maps to conditional waiting behaviour in the implementation. Our clause, $\mathbf{when}(g)\{A\}$, gives rise to the following fragment $\mathbf{while}(!TJG(g))\{wait();\}\ TJA(A);$ in the implementation. Here the built-in Java *wait* method is used to block entry to the conditional critical region, A, for as long as the condition for entry, g, 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. TJG and TJA are functions mapping the guard and action respectively to Java statements. Typically, operators of the OC-B guard require mapping to Java operators such equality, '=' in OC-B, maps to '==' in Java. In OC-B actions, ':=', the assignment operator maps to '='. An additional consideration is that a waiting thread may be interrupted; in this situation a Java *InterruptedException* is thrown, which must be caught by the waiting process. The code for handling this exception can be seen in Fig. 4 the example of translation of an OC-B specification to Java. We present a few example rules to illustrate the mapping but do not attempt a comprehensive treatment, we type translation functions TJA as,

Definition 7.1 $TJA \in Atomic \rightarrow JavaStatement$

An example translation rule is the mapping of a create call to Java code, where the new thread is started following its creation.

Definition 7.2 $TJA(v :=\ P.\mathbf{create}(ap_1, \ldots, ap_k))$
$=$
$v = \mathbf{new}\ P(ap_1, \ldots, ap_k);$
$\mathbf{new}\ Thread(v).start();$

where v is a variable of the caller which is assigned the new instance,
P is the type of class being instantiated,
and $ap_1, \ldots, ap_k$ are actual parameters.

The translation rule for the waiting construct follows, additional exception handling code is generated which can be seen in the example.

Definition 7.3 $TJA(\mathbf{when}(g)\{a\})$
$=$
$\mathbf{while}(!TJG(g))wait();$
$TJA(a);$

```
public class Prc implements java.lang.Runnable {
 private Shared shared ; private int i; private int count;

 public Prc(Shared shd){
   shared = shd ; i = 0 ; count = 0 ;
 }

 public void run(){
   while(count < 100){
     i = shared.getVal(); count = count+1;
   }
 }
}

public class Shared{
  private int val; private int attempts;

  public Shared(){ val = 0 ; attempts = 0 ;}

  public synchronized int getVal(){
    try{while(!(val > 0))wait();}catch(InterruptedException e){...}
    attempts = attempts + 1 ; return val ;
  }

  public synchronized void setVal(int v){ val = v ;}
}

public class Main{
 private static Prc pr1; private static Prc pr2;
 private static Shared sh ; ...

 public static void main(String[] args){
   sh = new Shared();
   pr1 = new Prc(sh);
   new Thread(pr1).start();
   pr2 = new Prc(sh);
   new Thread(pr2).start(); ...
 }
}
```

Fig. 4. Example of Translation to Java Code

8 Related Work

Related work is that of JCSProB, described in [30], which makes use of the JCSP libraries. JCSP [28,29] establishes a link between CSP [9,19] and Java. The JCSP libraries provide an implementation of the *Occam* concurrency framework, it uses a message passing, rendezvous style, as a basis for communication between concurrent

Java threads. Using JCSProB the ProB [14] tool can be used to construct and model check a combined CSP specification and B machine, which can then be translated to Java code. Our work is an alternative to this style and uses a shared memory approach, where processes share data in memory and accesses are protected using synchronized method calls. We also tailor our approach to the new Event-B tool rather than classical B.

UML-B and the U2B translator [10,16,21,22,23,25] established a basis for specifying B developments using a UML modelling tool, an updated version [24] is available as a plug-in compatible with the latest Event-B tool. We use some of the concepts described in UML-B to model objects and instances, but our notation introduces process classes that give rise to concurrently executing processes, with interleaving operations. The sequential operator used within a non-atomic operation defines points where interleaving may take place in addition to points we define in the looping and branching clauses. We define monitor classes that are shared between processes, and also define a mapping to Java code which is absent from UML-B. The OC-B syntax incorporates features such as the non-atomic looping and branching clauses which are not part of UML-B.

Object-Z [20] is a specification language which is an extension of the Z notation, it incorporates the notion of classes. A class schema encapsulates the state and behaviour of a class, and variables can take the type of a class. Inheritance mechanisms are used to clarify the structure of the systems and aid refinement and verification. Object-Z differs from OCB in a number of ways, for example we do not incorporate the notion of inheritance and we do not intend to refine an OCB specification. OCB forms a link in the development process between the Event-B modelling language and the implementation, Object-Z is used for system specification.

VDM++ [5] is an object oriented approach which is an extension of VDM-SL [11], UML diagrams are used to specify an object oriented development which are mapped to an underlying VDM++ model. VDM++ can be translated to Java but is not able to model features involving concurrency. Circus combines CSP [9,19] and Z [26]. The JCircus [7] translation tool gives rise to Java code which is intended to serve as an animator for circus. JCircus makes use of the JCSP libraries and gives rise to Java code that is based on the message passing approach, in this respect it is similar to JCSProB.

9 Conclusions and Future Work

Our work shows how to link an Event-B model to an object-oriented implementation by means of an intermediate specification using the OC-B notation; to our knowledge there is no other, similar approach to linking the two. The notation incorporates concurrent aspects of the implementation, allowing specification of monitor classes with atomic procedure definitions; and process classes with non-atomic operations and atomic procedure calls. We have defined a mapping from OC-B to Event-B and shown some of the rules here, we have also presented an example refinement and translation of an OC-B specification to an Event-B model and Java code. We first showed the mapping to Event-B using guarded command language syntax. We introduced the notion of processes with non-atomic operations,

consisting of labelled atomic clauses. The labels map to program counter values used in guards to model the order of execution; and the guarded actions of labelled atomic clauses map to guards and actions of an event. 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. Object-oriented features were then added; aspects of this relate to the underlying approach to modelling objects of UML-B. Mapping of variables was discussed; each variable belongs to a class and can be referred to in an OC-B clause (in a guard, action or expression). Due to the fact that we map to a model with instances, we require a translation function to map each occurrence of a variable in an OC-B clause, to a variable associated with a specific instance in the corresponding Event-B clause.

We introduced some of the syntactic sugar that provides a simple mapping to Java for the branching, looping and guarding (conditional waiting) constructs. It also fits with the object-oriented style, appropriate for the specification of implementation related details. We then presented an example abstraction, an OC-B specification for an implementation of the abstraction, and the translation of the OC-B specification to Event-B. The resulting Event-B model is a refinement of the abstract model. If required we could generate proof obligations to show that the refinement preserves some liveness properties. In this way it is possible to show that a refined model does not deadlock more frequently than its abstract counterpart (which is obviously not the case in our example), or indeed there may be a requirement to show that the system to be completely deadlock free. It is also possible to show that the new events of a refinement do not take over forever (divergence), which would prevent events of the abstract model from being enabled.

We find that the Event-B model arising from the translation of an OC-B specification 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 rationalise the approach, which could be achieved by the development of some patterns and guidelines, and maybe a calculus in the manner of Morgan's refinement calculus [18]. This will aid construction of OC-B specifications from Event-B models. The issue of modularity can largely be addressed by approaches such as decomposition [4] of the Event-B model itself. Our contribution is mainly to understand how to link Event-B with object-orient concepts that incorporate concurrency at a useful level of atomicity. This has been done with a view to producing a refinement of some more abstract model, rather than using the OC-B specification as a major part of the development process; for this reason we do not consider structuring mechanisms to be an important feature at this stage, however it will be useful to investigate this in the future.

The mapping to Java was then discussed, and an example of the resulting Java code presented. The OC-B 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.

We have developed prototype tool support for our approach, integrating with the RODIN Event-B tool. It is based on the Eclipse Platform [27] and incorporates an Eclipse based utility for construction of OC-B specifications. We have developed plug-ins to translate OC-B specifications to Event-B and Java source code. We have used the tool development to gain insight, and validate our theoretical work. We also hope the tool will be a useful legacy to be extended further with more useful features in the future.

In future work we plan to introduce transactional constructs. These will allow access to multiple shared objects, and will allow us to remove some of the restrictions in place at the moment. We plan to use the *java.utils.concurrent* packages for greater efficiency and flexibility, for instance techniques can be applied to overcome the nested monitor problem by controlling lock acquisition and release. We believe that specification using OC-B can ease the transition between formal modelling, at an abstract level, and providing a concurrent implementation. Reasoning about concurrency is simplified by abstracting away details of locking, and by providing a clear view of atomicity.

References

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

[2] J.R. Abrial. Event based sequential program development: Application to constructing a pointer program. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, 2003.

[3] 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.

[4] J.R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inform.*, 77(1-2):1–28, 2007.

[5] CSK Systems Corporation. The vdm++ language manual.

[6] 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.

[7] A. Freitas and A. Cavalcanti. Automatic translation from *ircus* 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.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification - Third Edition*. Addison-Wesley, 2004.

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

[10] 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.

[11] C.B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[12] 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.

[13] D. Lea. *Concurrent Programming in Java (Second Edition): Design Principles and Patterns*. Addison-Wesley, 2004.

[14] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *Proceedings of Formal Methods Europe 2003*, 2003.

[15] A.M. Lister. The problem of nested monitor calls. *Operating Systems Review*, 11(3):5–7, 1977.

[16] J.P. Mermet, editor. *UML-B Specification for Proven Embedded Systems Design*. Kluwer, 2004.

[17] C. Morgan. Procedures, parameters, and abstraction: separate concerns. *Sci. Comput. Program.*, 11(1):17–27, 1988.

[18] C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, June 1994.

[19] 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.

[20] G. Smith. *The Object-Z specification language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[21] 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.

[22] 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.

[23] C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 2006.

[24] 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.

[25] C. Snook, M. Butler, and I. Oliver. Towards a UML profile for UML-B. Technical report, Electronics and Computer Science, University of Southampton, 2003.

[26] J. M. Spivey. Understanding Z: A specification language and its formal semantics. *Cambridge Tracts in Theoretical Computer Science*, 3, 1988.

[27] The Eclipse Project. Eclipse - an Open Development Platform. Available at http://www.eclipse.org/.

[28] P.H. Welch and J.M.R. Martin. A CSP model for Java multithreading. In *Software Engineering for Parallel and Distributed Systems*, 2000.

[29] 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.

[30] L. Yang and M. Poppleton. Automatic translation from combined and csp specification to java programs. In J. Julliand and O. Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2007.