Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 201 (2008) 45–67
www.elsevier.com/locate/entcs

Towards the Formal Verification of a Java Processor in Event-B

Neil Evans and Neil Grant

AWE, Aldermaston, UK.

Abstract

Formal verification is becoming more and more important in the production of high integrity microprocessors. The general purpose formal method called Event-B is the latest incarnation of the B Method: it is a proof-based approach with a formal notation and refinement technique for modelling and verifying systems. Refinement enables implementation-level features to be proven correct with respect to an abstract specification of the system. In this paper we demonstrate an initial attempt to model and verify Sandia National Laboratories' Score processor using Event-B. The processor is an (almost complete) implementation of a Java Virtual Machine in hardware. Thus, refinement-based verification of the Score processor begins with a formal specification of Java bytecode. Traditionally, B has been directed at the formal development of software systems. The use of B in hardware verification could provide a means of developing combined software/hardware systems, i.e. codesign.

Keywords: Java processor, microcoded architecture, Event-B, refinement

1 Introduction

The Score processor has been designed at Sandia National Laboratories in the United States to be used as an embedded target for use with their components modelling system (called Advanced System Simulation Emulation and Test, or 'ASSET'). Now in its second generation, the processor is a hardware implementation of an almost complete Java Virtual Machine. In fact, the implementation far exceeds Sun's expectation of an embedded target. The SSP (Sandia Secure Processor) project started ten years ago, and the SSP2 (now called the Scalable Core Processor, or 'Score') is the current design. The redesign has allowed the processor architecture to be simplified, and this along with implementation efficiencies has allowed significantly more functionality.

doi:10.1016/j.entcs.2008.02.015

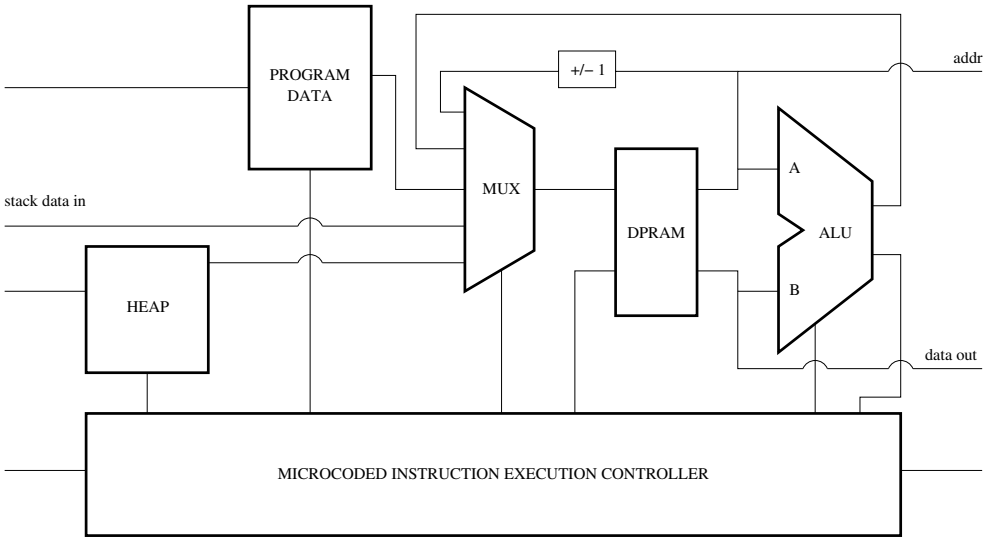

Fig. 1. An Abstract Score Architecture

The ASSET toolset is written in Java and uses Java to describe the component behaviour; this Java code can be compiled without modification to work on the Score processor.

Currently, Sandia uses the following (informal) validation checks on the Score processor:

- ring fencing (monitoring runtime memory access) in Java to check that opcodes do not do anything outside their remit;
- internal consistency checks (by the class loader) and a tree equivalence check;
- regression testing;
- comparison tests between two independent models - one in Java and the other in the hardware description language VHDL.

The motivation for this paper is to demonstrate initial results from an ongoing collaboration between AWE and Sandia to model and verify the Score processor using an established formal method. We choose the B Method, in particular the Event-B subset, for this purpose because it is a method with exceptional tool support which incorporates a dedicated refinement technique. We aim to prove that bytecodes are correctly implemented by microcode instructions.

Figure 1 shows a simplified architecture of the Score processor. The specific

details of the architecture are not important for the purposes of this paper. When the Score gets a Java bytecode from the program memory interface it is translated into a sequence of microcode instructions from the microcode table (held in the Microcoded Instruction Execution Controller). The power and flexibility of Score comes from the use of a complex microcode table, which can be modified even after the processor has been put onto silicon. In fact, the microcode table can be tailored to contain only the required microcode. The current optimised microcode table (including all the currently supported JVM functionality) is only just over 1600 lines. The original Score processor had a hand-crafted microcode table that was impossible to maintain by anyone other than its creators. Now a systematic methodology takes a structured design for the code and compiles it into a table. Logical names replace numerical values and the microcode is built up from defined fields which are typechecked during compilation.

Sandia's approach allows customisations to be made based on required functionality or runtime requirements. The class loader can determine which bytecodes are used (and hence required) for a particular application, and all the other bytecodes can then be removed from the microcode specification. This allows the table to be reduced to a minimum if necessary. The microcode table flexibility allows the SSP structure to be used more generally than just for the JVM. Non-Java bytecode could also be interpreted on the processor, for example, to emulate another processor.

It is clear from Figure 1 that the microcode is largely responsible for the activities of the processor, although the arithmetic logic unit (ALU) is not transparent: it contains registers that are not under the control of the microcode. The program and heap memories are both 8-bit. However the JVM specification demands a 32-bit stack. The original SSP had an internal 1000 level 32-bit stack, but this was over-specified as typically only 32 levels were ever used. The stack is held in memory that is external to the processor. Within the processor, the state variable memory is a dual port RAM (DPRAM). It stores values and constants including temporary variables and values that represent the stack boundaries.

The next section gives an overview of the Event-B language and its notion of refinement. This is followed by a demonstration of our approach via an example analysis of the JVM instruction *iadd*. Our approach is then put into context with other formal approaches, after which we draw some conclusions. We also discuss how this work could fit in with another AWE-funded project to produce formally verified hardware. This would address the issue of proving correctness with respect to actual (clocked) hardware. It is hoped that the results presented here can be generalised to support the entire lifecycle of

hardware development and verification. The longevity of the B Method gives us confidence that well-maintained tool support will be available in the future.

2 Event-B

An abstract Event-B specification [9] comprises a static part called the *context*, and a dynamic part called the *machine*. The machine has access to the context via a **SEES** relationship. This means that all sets, constants, and their properties defined in the context are visible to the machine. To model the dynamic aspects, the machine contains a declaration of all of the state variables. The values of the variables are set up using the **INITIALISATION** clause, and values can be changed via the execution of *events*. Ultimately, we aim to prove properties of the specification, and these properties are made explicit using the **INVARIANT** clause in the machine. The tool support generates the proof obligations which must be discharged to verify that the invariant is maintained. It also has interactive and automated theorem proving capabilities with which to discharge the generated proof obligations.

Events are specialised B operations [1]. In general, an event E is of the form

$$E \;\widehat{=}\; \textbf{WHEN}\ G(v)\ \textbf{THEN}\ S(v)\ \textbf{END}$$

where $G(v)$ is a Boolean guard and $S(v)$ is a generalised substitution (both of which may be dependent on one or more state variables denoted by v) [1]. The guard must hold for the substitution to be performed (otherwise the event is *blocked*). There are three kinds of generalised substitution: *deterministic*, *empty*, and *non-deterministic*. The deterministic substitution of a state variable x is an assignment of the form $x := E(v)$, for expression E (which may depend on the values of state variables — including x itself), and the empty substitution is *skip*. The non-deterministic substitution of x is defined as

$$\textbf{ANY}\ t\ \textbf{WHERE}\ P(t, v)\ \textbf{THEN}\ x := F(t, v)\ \textbf{END}$$

Here, t is a local variable that is assigned non-deterministically according to the predicate P, and its value is used in the assignment to x via the expression F.

3 Refinement in Event-B

In order to express the desired properties of a system as succinctly as possible, an abstract specification will dispense with many of the implementation

[1] The guard is omitted if it is trivially true.

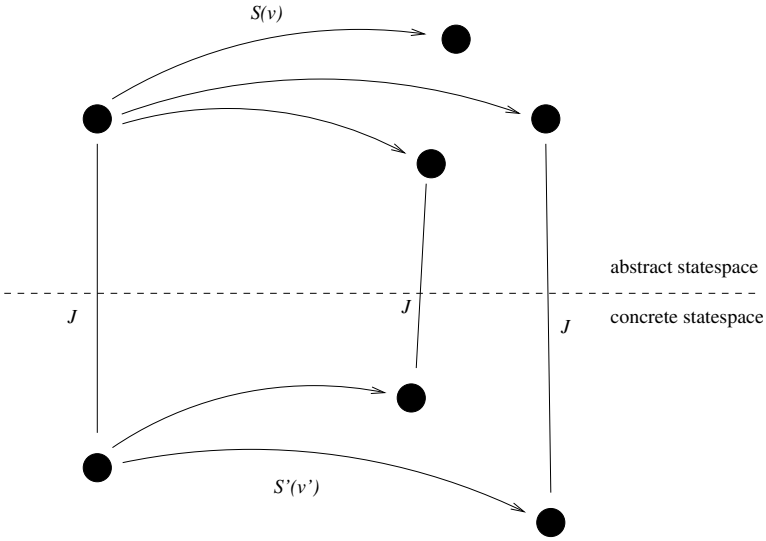

Fig. 2. Refinement of an Existing Event

details in favour of a more mathematical representation. Refinement is the means by which the artefacts of an implementation can be incorporated into a formal specification whilst maintaining the correct behaviour of the abstract specification. A demonstration of Event-B refinement will be given in the next section.

Traditionally, two main kinds of refinement are identified: *data refinement* and *operational refinement*. In data refinement, the aim is to replace abstract state with a more concrete, implementation-like state. Operation refinement aims to replace abstract algorithms (events) comprising abstract constructs with more program-like constructs. Operational refinement addresses the refinement of existing events. Refinement in Event-B also allows the introduction of new events. In many of his talks, Abrial gives a useful analogy for this form of refinement: an abstract specification is comparable to viewing a landscape from a great height. At this level of abstraction we get a good overview of the system without seeing many specific details. Refinement by introducing new events corresponds to moving closer to the ground: fine details that were previously out of sight are now revealed.

The context and machine of an abstract Event-B specification can be refined separately. Refinement of a context consists of adding additional sets, constants or properties (the sets, constants and properties of the abstract context are retained). The link between an abstract machine and its refinement

is achieved via a *gluing invariant* defined in the concrete machine. The gluing invariant relates concrete variables to those of the abstract model. Proof obligations are generated to ensure that this invariant is maintained.

The refinement of an existing event is depicted in Figure 2. If, in a state satisfying the gluing invariant J, a concrete event with (refined) generalised substitution S' and variable v' causes a transition to a new state, then the new state is related (via J) to a new state in the abstract world (i.e. a state resulting from the abstract event with generalised substitution S with abstract variable v). Note, the multiple arrows in the diagram indicate that generalised substitutions can be non-deterministic. Also note that it is not necessary for transitions in the abstract world to correspond to transitions in the concrete world (i.e. refinement can reduce the non-determinism).

New events introduced during Event-B refinement are allowed on the proviso that they cannot diverge (i.e. execute forever). This is necessary to ensure that new events cannot take control of the machine, thereby maintaining the visibility of existing events. More formally, divergence freedom is achieved by defining a variant which strictly decreases with the execution of each internal event. Since the variant is a natural number, the execution of internal events must eventually terminate to allow the execution of one or more existing events (after which internal activity may resume).[2] Of course, the desired properties of newly introduced events can be incorporated into the gluing invariant, and a proof that these properties are maintained is required.

4 Example Bytecode: *iadd*

To illustrate our approach using Event-B, we present the arithmetic operation *iadd* which pops two numbers off the stack, adds them together, and then pushes the result on to the stack. This example presents the kind of analysis that would be undertaken for all arithmetic and logical bytecode operations because, in all cases, operands are popped off the stack and the result is pushed back onto the stack. In the interest of simplicity, we only consider the effect of the operation on the data path. For example, we do not model the program counter, nor do we consider how the instruction gets called. In addition, we assume the operands are put on the stack by other instructions that are not considered here.

We begin by specifying the behaviour of *iadd* at a level of abstraction that is independent of the microcode that implements it. This level of abstraction

[2] Since the concrete events only operate on the state variables of the refined model, this form of refinement corresponds to a normal B refinement in which the newly introduced events simply refine the abstract (empty) event *skip*.

CONTEXT *STACK*
SETS
Stack ;
Bytecode ;
Status = { *ACTIVE* , *INACTIVE* }
CONSTANTS
iadd ,
null ,
cons ,
hd ,
tl ,
len
AXIOMS
$iadd \in Bytecode \land$
$null \in Stack \land$
$cons \in \mathbb{N} \times Stack \rightarrow (Stack - \{ null \}) \land$
$len \in Stack \rightarrow \mathbb{N} \land$
$hd \in (Stack - \{ null \}) \rightarrow \mathbb{N} \land$
$tl \in (Stack - \{ null \}) \rightarrow Stack \land$
$\forall n . (n \in \mathbb{N} \Rightarrow \forall s . (s \in Stack \Rightarrow hd (cons (n , s)) = n)) \land$
$\forall n . (n \in \mathbb{N} \Rightarrow \forall s . (s \in Stack \Rightarrow tl (cons (n , s)) = s)) \land$
$len (null) = 0 \land$
$\forall n . (n \in \mathbb{N} \Rightarrow \forall s . (s \in Stack \Rightarrow len (cons (n , s)) = 1 + len (s)))$
END

Fig. 3. Abstract stack context

is called the Instruction Set Architecture level (or ISA level). First we define an Event-B context to capture the properties of a stack (of type $\mathbb{N}$). This consists of a static definition of a list and its associated functions (i.e. we define a list as an abstract datatype). This is shown in Figure 3.

In order to specify a stack, we define a deferred set *Stack* and five constants: *null* denotes the empty stack, *cons* produces a new (non-empty) stack by putting an element at the top of an existing stack, *hd* returns the top element of a stack, *tl* returns a stack minus its top element, and *len* returns the length of a stack. The behaviours of these functions are defined as axioms. Note that *hd* and *tl* are partial functions with respect to the set *Stack* because $hd(null)$ and $tl(null)$ are undefined. In general, it may also be necessary to define an induction axiom to prove properties of a stack. However, this is not required here.

In addition to *Stack*, we have defined two further sets: *Bytecode* and *Status*. At this stage, only one element of *Bytecode* is declared, namely *iadd*. Further elements can be added via context refinement when necessary. The set *Status* (and its two elements *ACTIVE* and *INACTIVE*) is introduced as a consequence of Event-B refinement. We will see below why this set is necessary.

Next we define the dynamic behaviour as an Event-B machine. This is shown in Figure 4. This machine has access to the static elements via the

SEES clause. Three variables are defined: *opcode* holds the current JVM instruction, *stack* holds the current state of the stack, and *iadd_status* says whether the execution of *iadd* is in progress (i.e. *ACTIVE*) or not (i.e. *INACTIVE*). The implication statement in the machine's invariant says that there are enough elements on the stack whenever *iadd_status* is *ACTIVE*. The guards of the events guarantee this, but in the real world some other mechanism would be needed to ensure this. (It is the job of the class loader to prevent underflow of the stack.)

The variable *iadd_status* is introduced in anticipation of refinement. This is also the reason for two events: **iAdd_ini** activates the execution (but only when there are enough elements in the stack), and **iAdd** performs the necessary state update. One could imagine an event that would capture the behaviour of *iadd* in one step, i.e.:

iAdd $\widehat{=}$
WHEN
$len\ (\ stack\ ) > 1 \wedge$
$opcode = iadd$
THEN
$stack := cons\ (\ hd\ (\ stack\ ) + hd\ (\ tl\ (\ stack\ )\ )\ ,\ tl\ (\ tl\ (\ stack\ )\ )\ )$
END

without the need for a status variable. However, events with nontrivial guards and generalised substitutions such as this serve two purposes: the guard says what should hold at the beginning of an execution, and the generalised substitution says what should hold at the end. A refinement that introduces new events would force us to choose between executing the existing event first (to exercise the guard at the appropriate place), or last (to position the generalised substitution appropriately). Since Event-B does not allow events to be split, we are forced to define (at least) two events: one with the nontrivial guard, and another with the generalised substitution. We will say more about this when we consider the refinement itself.

4.1 A Refined Model

The *iadd* operation is broken down into 13 microcoded instructions on the Score processor. An in-depth understanding of the Score processor and how the microcode assembler is structured would be required to fully appreciate the instructions that are used. However, since the aim of this paper is to demonstrate Event-B refinement in this context, we simplify things by break-

MACHINE *ISA*
SEES *STACK*
VARIABLES
opcode ,
iadd_status ,
stack
INVARIANT
$opcode \in Bytecode \wedge$
$iadd\_status \in Status \wedge$
$stack \in Stack \wedge$
$iadd\_status = ACTIVE \Rightarrow len\ (\ stack\ ) > 1$
INITIALISATION
$opcode :\in Bytecode \ \|$
$stack := null \ \|$
$iadd\_status := INACTIVE$
EVENTS

iAdd_ini $\widehat{=}$
WHEN
$iadd\_status = INACTIVE \wedge$
$len\ (\ stack\ ) > 1 \wedge$
$opcode = iadd$
THEN
$iadd\_status := ACTIVE$
END ;

iAdd $\widehat{=}$
WHEN
$iadd\_status = ACTIVE$
THEN
$stack := cons\ (\ hd\ (\ stack\ ) + hd\ (\ tl\ (\ stack\ )\ )\ ,\ tl\ (\ tl\ (\ stack\ )\ )\ ) \ \|$
$iadd\_status := INACTIVE$
END

END

Fig. 4. Abstract machine for *iadd*

VARIABLES	**INVARIANT**
$opcode1$,	$opcode1 \in Bytecode \wedge$
$iadd\_status1$,	$iadd\_status1 \in Status \wedge$
SP ,	$SP \in \mathbb{N} \wedge$
$stack1$,	$stack1 \in \mathbb{N}_1 \pfun \mathbb{N} \wedge$
$stackDataIn$,	$stackDataIn \in \mathbb{N} \wedge$
$ALURegA$,	$ALURegA \in \mathbb{N} \wedge$
$ALURegB$,	$ALURegB \in \mathbb{N} \wedge$
$ALUOutReg$,	$ALUOutReg \in \mathbb{N} \wedge$
$stackDataOut$,	$stackDataOut \in \mathbb{N} \wedge$
$stackDataInSet$,	$stackDataInSet \in BOOL \wedge$
$ALURegASet$,	$ALURegASet \in BOOL \wedge$
$ALURegBSet$,	$ALURegBSet \in BOOL \wedge$
$ALUOutSet$,	$ALUOutSet \in BOOL \wedge$
$stackDataOutSet$	$stackDataOutSet \in BOOL$

Fig. 5. Refined state variables and their types

ing the *iadd* operation into 7 *pseudo*-microcoded instructions. This description ignores some features of the processor, but it still incorporates many of the actual implementation details. This compromise allows us to demonstrate the refinement technique involved. By proving this lower-level model is an Event-B refinement of the abstract model, we demonstrate that the low-level behaviour is faithful to the ISA specification of *iadd*. Although we only present one refinement here, the approach is similar for all bytecodes.

The context of the refined model remains the same as the abstract model, so we begin by listing the variables and their associated types in the refined machine. This is shown in Figure 5. Three of the variables have counterparts in the abstract model: $opcode1$, $iadd\_status1$ and $stack1$. Of these, $stack1$ is most interesting because it is refined by replacing the abstract datatype *Stack* with a partial function mapping (positive) natural numbers to natural numbers. This is closer to the real implementation of a stack because we can think of the domain of the function as memory addresses and the range as the contents of the memory. The variable SP is introduced to represent a pointer to the head of the stack.

The other variables introduced here model the registers involved in the computation: $stackDataIn$ and $stackDataOut$ hold values in transit from/to the stack, and $ALURegA$, $ALURegB$ and $ALUOutReg$ hold values entering and leaving the ALU. The remaining variables are Boolean flags that are needed to record the state of the registers. They do not correspond to actual components on the Score processor, but they are needed to guard the events

so that they are called at the appropriate time.

The invariant shown in Figure 5 only gives the types of the state variables. It says nothing about the correspondence between the concrete variables and the abstract variables. We shall derive the necessary clauses in a systematic way after we have introduced the events. First we consider the refinements of the existing events. These are shown in Figure 6. The event **iAdd_ini** is almost identical to its counterpart in Figure 4, except we use the concrete variables $iadd\_status1$ and $opcode1$, and the conjunct $SP > 1$ replaces $len(stack) > 1$. This will impose conditions on the gluing invariant when it is derived. In addition to an *ACTIVE* status, the guard of the refined event **iAdd** now depends on the variable *stackDataOutSet*. This is necessary to block the event until a meaningful value is ready to be pushed onto the stack (which is achieved by the assignment in the generalised substitution). Since the event completes the computation for *iadd*, the flags are reset in preparation for the next arithmetic operation.

The events introduced in this refinement are responsible for updating the state variables so that, when the event **iAdd** executes, $stack1$, SP and *stackDataOut* hold the correct values to fulfil the requirements of *iadd*. This happens in a number of stages, which are summarised below:

- **readStackAndDec** pops an element off the stack, decreases the stack pointer and sets *stackDataInSet* to indicate that *stackDataIn* holds a value to be added;
- **writeALURegA** takes the value stored in *stackDataIn* and passes it to the ALU register *ALURegA*, and a flag is set to indicate this;
- **writeALURegB** takes the value stored in *stackDataIn* and passes it to the ALU register *ALURegB*, and a flag is set to indicate this;
- **ALUAdd** adds the values of the two ALU registers and assigns this to *ALUOutReg*;
- **incRegAndLoadStack** assigns *ALUOutReg* to *stackDataOut* in readiness for the stack to be updated.

Note that the order of the events is implicit, and depends on the truth values of the guards. In this case, **readStackAndDec** will occur twice in every execution of *iadd* in order to assign the two input registers of the ALU. The complete set of definitions for the new events is shown in Figure 7.

4.2 Constructing a Gluing Invariant

Ultimately, our gluing invariant should relate the abstract variable *stack* with the concrete variable $stack1$ (i.e. they should be equivalent in some sense).

iAdd_ini $\widehat{=}$
WHEN
$iadd\_status1 = INACTIVE \wedge$
$SP > 1 \wedge$
$opcode1 = iadd$
THEN
$iadd\_status1 := ACTIVE$
END

iAdd $\widehat{=}$
WHEN
$iadd\_status1 = ACTIVE \wedge$
$stackDataOutSet = TRUE$
THEN
$stack1 := stack1 \cup \{ SP \mapsto stackDataOut \} \parallel$
$iadd\_status1 := INACTIVE \parallel$
$stackDataOutSet := FALSE \parallel$
$ALURegASet := FALSE \parallel$
$ALURegBSet := FALSE \parallel$
$ALUOutSet := FALSE$
END

Fig. 6. Refining the existing events

readStackAndDec $\widehat{=}$
WHEN
$iadd\_status1 = ACTIVE \wedge$
$SP \in \mathsf{dom}(\ stack1\ ) \wedge$
$stackDataInSet = FALSE \wedge$
$(\ ALURegASet = FALSE \vee$
$ALURegBSet = FALSE\ )$
THEN
$SP := SP - 1 \parallel$
$stackDataIn := stack1\ (\ SP\ ) \parallel$
$stack1 := \{\ SP\ \} \lhd stack1 \parallel$
$stackDataInSet := TRUE$
END

writeALURegB $\widehat{=}$
WHEN
$stackDataInSet = TRUE \wedge$
$ALURegBSet = FALSE$
THEN
$ALURegB := stackDataIn \parallel$
$stackDataInSet := FALSE \parallel$
$ALURegBSet := TRUE$
END

writeALURegA $\widehat{=}$
WHEN
$stackDataInSet = TRUE \wedge$
$ALURegASet = FALSE$
THEN
$ALURegA := stackDataIn \parallel$
$stackDataInSet := FALSE \parallel$
$ALURegASet := TRUE$
END

ALUAdd $\widehat{=}$
WHEN
$ALURegASet = TRUE \wedge$
$ALURegBSet = TRUE \wedge$
$ALUOutSet = FALSE$
THEN
$ALUOutReg :=$
$ALURegA + ALURegB \parallel$
$ALUOutSet := TRUE$
END

incRegAndLoadStack $\widehat{=}$
WHEN
$ALUOutSet = TRUE \wedge$
$stackDataOutSet = FALSE$
THEN
$SP := SP + 1 \parallel$
$stackDataOut := ALUOutReg \parallel$
$stackDataOutSet := TRUE$
END

Fig. 7. Introducing new events

However, before we do this it is necessary to address a number of proof obligations that arise in the refined model. These concern the guards of the events **iAdd_ini** and **iAdd**. The theory underlying the B Method dictates that the guards of refined events must be at least as strong as the guards that they refine. In the case of **iAdd_ini**, we have to prove:

$$(iadd\_status1 = INACTIVE \ \wedge \ SP > 1 \ \wedge \ opcode1 = iadd) \Rightarrow (iadd\_status = INACTIVE \ \wedge \ len(stack) > 1 \ \wedge \ opcode = iadd)$$

and in the case of **iAdd**, we have to prove:

$$(iadd\_status1 = ACTIVE \wedge stackDataOutSet = TRUE) \Rightarrow iadd\_status = ACTIVE$$

One might be tempted to add the following clauses to the invariant:

$$(iadd\_status1 = INACTIVE \Rightarrow add\_status = INACTIVE) \wedge$$
$$(iadd\_status1 = ACTIVE \Rightarrow add\_status = ACTIVE) \wedge$$
$$SP = len(stack) \wedge$$
$$opcode1 = opcode$$

However, the clause $SP = len(stack)$ is not invariant because, in the concrete model, the value of SP changes prior to the execution of **iAdd** whereas the length of *stack* remains the same until (the abstract) **iAdd** event is executed. This illustrates a key feature of the approach: relationships such as this are only relevant in certain states. In this case, we can weaken the clause as follows:

$$iadd\_status1 = INACTIVE \Rightarrow SP = len(stack)$$

The proof of this implication is nontrivial because, even though we are not interested in the active states, we have to analyse them in order to establish the final value of SP. Rather than demonstrating this here, we shall demonstrate a stronger property of the stack.

Our aim is to show that the concrete model captures the behaviour of the *iadd* operation. We do this with a refinement proof in Event-B. The aim, therefore, is to show that the stacks resulting from the computation in the concrete and abstract world are equivalent, on the assumption that the stacks were equivalent prior to the computation. This is depicted in Figure 8. The top half of the diagram represents the abstract world. Two stacks are shown: one prior to performing *iadd*, and one after. They are connected by two transition arrows and one intermediate (active) state. The leftmost element in both stacks is the head. The bottom half of the diagram gives a concrete representation of the same situation (in this case, the stacks are made up of index/value pairs). Here, the bottom pair is the head, and is labelled by the pointer SP. Note that there are more transitions and intermediate states involved in the concrete world.

We define a predicate *eqv* to capture the relationship between these two viewpoints (this is indicated by the vertical lines in the diagram). We begin

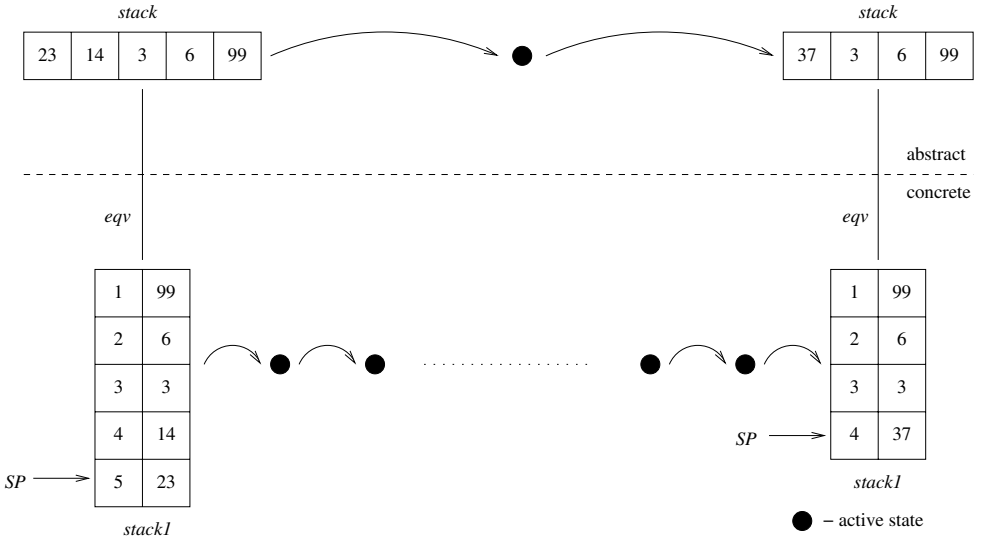

Fig. 8. Relating inactive states

by declaring the relationship for inactive states. If *eqv* is of type $\mathbb{N} \times (\ \mathbb{N}_1 \nrightarrow \mathbb{N}\ ) \times Stack \rightarrow BOOL$, then this is written formally (and added to the invariant) as:

$$iadd\_status1 = INACTIVE \quad \Rightarrow \quad eqv(SP, stack1, stack)$$

where the (well-founded) definition of *eqv* is as follows:

$$\begin{array}{l} eqv(n, s, null) \quad = \quad n = 0 \\ eqv(n, s, cons(h, t)) \quad = \quad n > 0 \ \wedge \ n \in \mathsf{dom}\ (\ s\ )\ \wedge\ s\ (\ n\ ) = h\ \wedge \\ eqv(n-1, s, t) \end{array}$$

When applied in the invariant, this definition ensures that the elements of a (non-*null*) abstract stack correspond to the entries in the concrete stack from index *SP* down to index 1, and *SP* is 0 when the abstract stack is empty.

Since the concrete event **iAdd** yields a state in which $iadd\_status1$ is $INACTIVE$, the following proof obligation is generated by the tool:

$$eqv(SP,\ stack1 \cup \{SP \mapsto stackDataOut\},\ cons(hd(stack) + hd(tl(stack)), tl(tl(stack))))$$

This is due to the generalised substitution arising in the **iAdd** event defined in the refined model:

iAdd $\widehat{=}$
WHEN
$iadd\_status1 = ACTIVE \wedge$
$stackDataOutSet = TRUE$
THEN
$stack1 := stack1 \cup \{ SP \mapsto stackDataOut \} \ \|$
$iadd\_status1 := INACTIVE \ \|$
$\vdots$
END

and the corresponding definition in the abstract model:

iAdd $\widehat{=}$
WHEN
$iadd\_status = ACTIVE$
THEN
$stack := cons\ (\ hd\ (\ stack\ ) + hd\ (\ tl\ (\ stack\ )\ )\ ,\ tl\ (\ tl\ (\ stack\ )\ )\ ) \ \|$
$iadd\_status := INACTIVE$
END

It is necessary to prove (or *discharge*) such proof obligations in order to demonstrate that the invariant is maintained. The above proof obligation is true if, prior to the **iAdd** event, the following three subgoals can be proven:

- SP is the 'next' unoccupied position in $stack1$;
- $stackDataOut = hd(stack) + hd(tl(stack))$, i.e. the sum of the first two elements of $stack$;
- $eqv(SP - 1,\ stack1,\ tl(tl(stack)))$, i.e. $stack1$ is equivalent to the abstract stack minus its top two elements.

At this stage it is impossible to confirm or refute these subgoals. We have to consider the sequence of events that would have led up to the occurrence **iAdd** in the concrete model. Our approach is to augment the refinement's invariant with any proof obligations that cannot be proven. Then we use the tool to generated additional proof obligations. This process is repeated until no further proof obligations are generated. We begin by adding the above proof obligation to the invariant. However, we do this under the assumption that the guard of **iAdd** (otherwise the event would not have occurred). In particular, we assume $stackDataOutSet = TRUE$:

$$stackDataOutSet = TRUE \ \Rightarrow$$
$$eqv(SP,\ stack1 \cup \{SP \mapsto stackDataOut\},\ cons(hd(stack) + hd(tl(stack)), tl(tl(stack))))$$

As a consequence, the proof obligation now disappears and a new proof obligation is generated instead:

$$eqv(SP+1, stack1 \cup \{SP+1 \mapsto ALUOutReg\},\ cons(hd(stack)+hd(tl(stack)), tl(tl(stack))))$$

This is due to the event **incRegAndLoadStack**, because its generalised substitution is responsible for setting $stackDataOutSet$ to $TRUE$:

incRegAndLoadStack $\widehat{=}$
WHEN
$ALUOutSet = TRUE \wedge$
$stackDataOutSet = FALSE$
THEN
$SP := SP + 1 \ \|$
$stackDataOut := ALUOutReg \ \|$
$stackDataOutSet := TRUE$
END

By performing the substitution on the subgoals, we can derive subgoals that are sufficient to prove this newly generated proof obligation:

- $SP + 1$ is the 'next' unoccupied position in $stack1$;
- $ALUOutReg = hd(stack) + hd(tl(stack))$;
- $eqv((SP + 1) - 1,\ stack1,\ tl(tl(stack)))$.

Of course, the last proof goal simplifies to:

- $eqv(SP,\ stack1,\ tl(tl(stack)))$.

Once again, we cannot confirm or refute these, so we add the generated proof obligation to the invariant. This time, we assume that the guard of **incRegAndLoadStack** holds:

$$ALUOutSet = TRUE \ \wedge \ stackDataOutSet = FALSE \ \Rightarrow$$
$$eqv(SP+1, stack1 \cup \{SP+1 \mapsto ALUOutReg\},\ cons(hd(stack)+hd(tl(stack)), tl(tl(stack))))$$

As before, the proof obligation is now replaced by a new proof obligation, this time arising from **ALUAdd** (because this event assigns $TRUE$ to $ALUOutSet$):

$$eqv(SP+1, stack1 \cup \{SP+1 \mapsto ALURegA + ALURegB\},\ cons(hd(stack)+hd(tl(stack)), tl(tl(stack))))$$

This proof obligation differs from the previous one because $ALUOutReg$ is assigned to be the sum of $ALURegA$ and $ALURegB$:

ALUAdd $\widehat{=}$
WHEN
$ALURegASet = TRUE \wedge$
$ALURegBSet = TRUE \wedge$
$ALUOutSet = FALSE$
THEN
$ALUOutReg := ALURegA + ALURegB \ \|$
$ALUOutSet := TRUE$
END

The second of the three subgoals is affected by the generalised substitution in this event:

- $ALURegA + ALURegB = hd(stack) + hd(tl(stack))$,

We are required to look further to discover the values assigned to $ALURegA$ and $ALURegB$. First we add the proof obligation to the invariant:

$$ALURegASet = TRUE \ \wedge \ ALURegBSet = TRUE \ \wedge \ ALUOutSet = FALSE \ \Rightarrow$$
$$eqv(SP + 1, stack1 \cup \{SP + 1 \mapsto ALURegA + ALURegB\},$$
$$cons(hd(stack) + hd(tl(stack)), tl(tl(stack))))$$

This situation is a bit more interesting because there are two possible paths that could reach a state in which $ALURegASet$ and $ALURegBSet$ are true: (i) if $ALURegASet$ is true and an occurrence of **writeALURegB** sets $ALURegBSet$ to $TRUE$; (ii) if $ALURegBSet$ is true and an occurrence of **writeALURegA** sets $ALURegASet$ to $TRUE$.

writeALURegA $\widehat{=}$
WHEN
$stackDataInSet = TRUE \wedge$
$ALURegASet = FALSE$
THEN
$ALURegA := stackDataIn \ \|$
$stackDataInSet := FALSE \ \|$
$ALURegASet := TRUE$
END

writeALURegB $\widehat{=}$
WHEN
$stackDataInSet = TRUE \wedge$
$ALURegBSet = FALSE$
THEN
$ALURegB := stackDataIn \ \|$
$stackDataInSet := FALSE \ \|$
$ALURegBSet := TRUE$
END

Hence, two proof obligations are generated, which we add to the invariant

with the appropriate assumptions:

$$\begin{aligned}&\text{(i) } stackDataInSet = TRUE \ \wedge \ ALURegASet = TRUE \ \Rightarrow \\ &\qquad eqv(SP+1, stack1 \cup \{SP+1 \mapsto ALURegA + stackDataIn\}, \\ &\qquad\qquad cons(hd(stack) + hd(tl(stack)), tl(tl(stack))))\end{aligned}$$

which has a further impact on the second of the three subgoals:

- $ALURegA + stackDataIn = hd(stack) + hd(tl(stack))$,

$$\begin{aligned}&\text{(ii) } stackDataInSet = TRUE \ \wedge \ ALURegBSet = TRUE \ \Rightarrow \\ &\qquad eqv(SP+1, stack1 \cup \{SP+1 \mapsto stackDataIn + ALURegB\}, \\ &\qquad\qquad cons(hd(stack) + hd(tl(stack)), tl(tl(stack))))\end{aligned}$$

which also has an impact on the second of the three subgoals:

- $stackDataIn + ALURegB = hd(stack) + hd(tl(stack))$.

Note, the assumption $stackDataInSet = TRUE$ in (i) implies that $ALURegBSet$ is $FALSE$ (which is required to enable the guard of **writeALURegB**). Similarly, the assumption $stackDataInSet = TRUE$ in (ii) implies that $ALURegASet$ is $FALSE$.

The addition of the above implications to the invariant forces us to consider the behaviour of **readStackAndDec** which precedes the occurrences of both **writeALURegA** and **writeALURegB**. Every time this event occurs, it assigns the top element of the (concrete) stack to *stackDataIn* and decrements the stack pointer *SP*:

readStackAndDec $\widehat{=}$
WHEN
$iadd\_status1 = ACTIVE \wedge$
$SP \in \mathsf{dom}\ (\ stack1\ ) \wedge$
$stackDataInSet = FALSE \wedge$
$(\ ALURegASet = FALSE \vee$
$\quad ALURegBSet = FALSE\ )$
THEN
$SP := SP - 1 \ \parallel$
$stackDataIn := stack1\ (\ SP\ ) \ \parallel$
$stack1 := \{\ SP\ \} \lhd stack1 \ \parallel$
$stackDataInSet := TRUE$
END

This event occurs twice during each execution of the (concrete) *iadd* operation, so multiple cases have to be considered:

1. $stackDataInSet = FALSE \ \wedge \ ALURegASet = TRUE \ \wedge \ ALURegBSet = FALSE$.
   In this state, **readStackAndDec** is enabled to assign a value to $ALURegB$. It is sufficient to discharge the proof obligation generated in this state if we can prove the following subgoals:
   · $(SP - 1) + 1$ is the 'next' unoccupied position in $\{SP\} \lhd stack1$;
   · $ALURegA + stack1(SP) = hd(stack) + hd(tl(stack))$;
   · $eqv(SP - 1,\ \{SP\} \lhd stack1,\ tl(tl(stack)))$.
   The first of these proof goals simplifies to:
   · SP is the 'next' unoccupied position in $\{SP\} \lhd stack1$.
2. $stackDataInSet = FALSE \ \wedge \ ALURegASet = FALSE \ \wedge \ ALURegBSet = TRUE$.
   In this state, **readStackAndDec** is enabled to assign a value to $ALURegA$. It is sufficient to discharge the proof obligation generated in this state if we can prove the following subgoals:
   · SP is the 'next' unoccupied position in $\{SP\} \lhd stack1$;
   · $stack1(SP) + ALURegB = hd(stack) + hd(tl(stack))$;
   · $eqv(SP - 1,\ \{SP\} \lhd stack1,\ tl(tl(stack)))$.
3. $stackDataInSet = FALSE \ \wedge \ ALURegASet = FALSE \ \wedge \ ALURegBSet = FALSE$.
   In this state, **readStackAndDec** is enabled to assign a value (non-deterministically) to either $ALURegA$ or $ALURegB$. It is sufficient to discharge the proof obligation generated in this state if we can prove the following subgoals:
   · $SP - 1$ is the 'next' unoccupied position in $\{SP\} \lhd (\{SP - 1\} \lhd stack1)$;
   · $stack1(SP) + stack1(SP - 1) = hd(stack) + hd(tl(stack))$;
   · $eqv((SP - 1) - 1,\ \{SP\} \lhd (\{SP - 1\} \lhd stack1),\ tl(tl(stack)))$.
   The third subgoal can be simplified to:
   · $eqv(SP - 2,\ \{SP\} \lhd (\{SP - 1\} \lhd stack1),\ tl(tl(stack)))$.

Finally we are at a point where we can complete the proof. If we add the following clause to the invariant, then we can prove all of the subgoals:

$$stackDataInSet = FALSE \ \wedge \ ALURegASet = FALSE \ \wedge \ ALURegBSet = FALSE \ \Rightarrow$$
$$eqv(SP, stack1, stack)$$

For example, if $eqv(SP, stack1, stack)$ is true then removing two elements from both stacks results in equivalent stacks, i.e. $eqv(SP-2,\ \{SP\} \lhd (\{SP-1\} \lhd stack1),\ tl(tl(stack))))$, which confirms the third subgoal. Note that, by adding the final implication to the invariant, we have in fact weakened our original gluing invariant:

$$iadd\_status1 = INACTIVE \quad \Rightarrow \quad eqv(SP, stack1, stack)$$

In this analysis we have used the tool to generate the invariant for us. In discharging the proof obligations, all but two of the proof obligations were proven automatically. This kind of approach is not easy to follow when written down (even when it's simplified, as in the description above) so it is not very practical for hand-written proofs. However, the tool support that accompanies Event-B keeps track of all outstanding proof obligations, and provides an easy user interface and theorem proving support for interactive proofs.

4.3 Other Issues

One outstanding issue concerns parameters: the Score processor's microcoded instruction set includes instructions that take input parameters. Unlike operations in the B Method, events in Event-B do not allow input parameters. Instead, the **ANY** clause introduced in Section 2 can be used to model instruction parameters. In terms of proof, a non-deterministic substitution of the form:

$$\textbf{ANY}\ t\ \textbf{WHERE}\ \cdots$$

will typically generate proof obligations of the form:

$$\forall\, t.\ \cdots$$

That is, a proof must consider all possible instantiations of the parameters that are modelled by the local variable t.

5 Other Approaches

The most substantial body of work in this area to date has been done by Panagiotis Manolios. His technique for modelling and verifying hardware motivated the investigation undertaken in this paper. The general purpose theorem proving system ACL2 [2] provides the mechanical support for his approach. Lisp is used as the modelling language, in which models of a similar level of abstraction to our own are constructed. In particular, instruction set architecture (ISA) models and microarchitecture (MA) models are defined using Lisp primitives [7].

In order to prove a correspondence between an ISA model and an MA model, a *refinement map* from MA states to ISA states is constructed which, in essence, says how to view an MA state as an ISA state. Typically, the map will 'forget' some of the details of the MA state in order to recover a corresponding ISA state. If, using this mapping, it is possible to derive a well-founded equivalence bisimulation relation (see [7]) then the models can be seen to be equivalent. Note that this (equivalence) notion of refinement differs from that of Event-B because, in the latter case, the behaviours of the concrete model should be more constrained (or less non-deterministic) than the abstract model. However, there is a similarity between the two approaches because this notion of bisimulation only allows finite stuttering. This corresponds to Event-B's notion of divergence freedom: events introduced in a refinement (i.e. those events that are hidden at the abstract level) cannot take infinite control. Otherwise, this would correspond to infinite internal activity (i.e. infinite stuttering) at the abstract level.

To overcome the difficulties associated with using automated theorem provers (in particular, the level of interaction), Manolios has enlisted the help of the UCLID tool [12] which makes use of SAT solving technology and BDD's to prove refinements between models [8]. In a similar way, users of Event-B can call upon the model checking tool ProB [6] to provide more automated tool assistance in the development of Event-B models. It has the capability to animate specifications, analyse invariants, and check refinements of finite state models.

6 Conclusion

In this paper we have applied Event-B refinement to the verification of a Java processor. In particular, we have demonstrated a proof of an example bytecode with respect to its microcoded instruction implementation. We have chosen to use Event-B in this investigation because it has an off-the-shelf (and free) formal development tool with a dedicated refinement technique. Hence, our proposed approach has been tailored to make full use of the tool.

Of course, the process of verification must be repeated for each bytecode but, since the microcoded instructions will be used repeatedly, existing invariants (such as those derived in Section 4.2) can be reused in different contexts. Hence, we can expect subsequent proofs to be less time consuming.

The Event-B tool is being developed with extensibility in mind. The decision to use Eclipse as an environment for the tool is based on its plug-in capability. For example, in addition to ProB, other tools such as a UML to B translator [11] are being built to interact directly with the Event-B tool. This

will provide alternative 'front ends' to the tool to enable formal development via other more familiar notations. Hence, it is likely that tools such as these will play a part in the development of future hardware projects rather than for post hoc verification.

AWE has been involved in using formal methods in hardware development for the last 15 years, and is keen to investigate formal techniques to make the production of rigorous hardware achievable. For instance, the development of computerised control systems requires verified hardware. Since no commercial processors have been available to meet this requirement, in-house hardware has been developed. Some early work, in collaboration with B-Core (UK), added hardware component libraries and a VHDL hardware description language code generator to the B Toolkit [5]. All hardware specifications written using this approach (called B-VHDL) mimic the structure of traditional VHDL programs and, hence, give a very low-level view of a development. The work presented in this paper investigates the applicability of the latest B technologies at a higher level of abstraction. Currently, a collaboration between AWE and the University of Surrey is investigating routes from high-level specifications (such as those presented in this paper) down to clocked physical hardware. In addition to B, other formal notations such as CSP are being used to specify and refine combined software/hardware models.

Acknowledgement

The authors thank the anonymous referees for their insightful comments.

References

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

[2] ACL2, http://www.cs.utexas.edu/users/moore/acl2/.

[3] Atelier B, http://www.atelierb.societe.com.

[4] B Core (UK) Ltd, http://www.b-core.com.

[5] Ifill W., Sorensen I., Schneider S.: *The use of B to Specify, Design and Verify Hardware*. In *High Integrity Software*, Kluwer Academic Publishers, 2001.

[6] Leuschel M., Butler M.: *ProB: A Model Checker for B*, FME 2003: Formal Methods, LNCS 2805, Springer, 2003.

[7] Manolios P.: *Refinement and Theorem Proving*, International School on Formal Methods for the Design of Computer, Communication, and Software Systems: Hardware Verification, Springer, 2006.

[8] Manolios P., Srinivasan S.: *A Complete Compositional Framework for the Efficient Verification of Pipelined Machines*, ACM-IEEE International Conference on Computer Aided Design, 2005.

[9] Métayer C., Abrial J. R., Voisin L.: *Event-B Language*, RODIN deliverable 3.2, http://rodin.cs.ncl.ac.uk (2005).

[10] Schneider S.: *The B Method: An Introduction*, Palgrave (2001).

[11] Snook C., Butler M.: *UML-B: Formal Modeling and Design Aided by UML*, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 15, Issue 1, 2006.

[12] UCLID, http://www.cs.cmu.edu/~uclid/.