Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 214 (2008) 183–202

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Refinement for Pipelining in Event-B

Neil Evans[1]

AWE, Aldermaston, UK.

Abstract

The refinement of an implementation-independent specification of an instruction set to a simple pipelined architecture is presented to illustrate subtleties in the formal development and analysis of pipelined hardware from such specifications. The Event-B language and its tool support (the Eclipse-based Rodin platform) is used for this purpose. The example demonstrates that naïve use of Event-B's superposition refinement fails to expose all of the potential hazards in pipelining. This paper introduces a form of 'event merging' to complete the analysis.

Keywords: Refinement, Event-B, pipelined architecture.

1 Introduction

The Control Systems team at AWE is currently undertaking the formal analysis of a hardware implementation of a Java Virtual Machine (JVM). The intention is to analyse the behaviour of the processor with respect to the instruction set presented in the official JVM documentation [8]. Since the instruction set of the JVM comprises over 200 bytecodes, this paper will illustrate one of the intended approaches using a considerably smaller example. However, the example is sufficient to demonstrate potential hazards in the development of pipelined hardware through refinement.

The work presented in this paper uses a dedicated notation and refinement technique for modelling systems: Event-B [10]. This is in contrast to other approaches, such as [9], in which specialised theories are constructed within general-purpose theorem provers. The Event-B notation is based on

[1] Email: Neil.Evans@AWE.co.uk

doi:10.1016/j.entcs.2008.06.009

action systems, and incorporates a refinement technique principally based on *superposition refinement* [3]. Event-B is supported by the Eclipse-based Rodin tool [14].

The example used in this paper comes from [9]. In his work, Manolios develops theories in ACL2 [2] to implement a refinement technique based on *well-founded equivalence bisimulation* [12]. In this context, the ACL2 language (i.e. Common Lisp) is used to define state models and associated computation steps at two levels of abstraction. In order to show a refinement between the models, it is necessary to construct a *refinement map* between the states and show that, for any related states, the relationship is preserved by computations performed at the two levels of abstraction (up to *finite stuttering*).

Informally, the ACL2 and Event-B approaches described above are similar because the refinement map in ACL2 corresponds to the *gluing invariant* in Event-B. A gluing invariant relates states in a refined (concrete) Event-B model with states in the corresponding abstract model. Finite stuttering in the ACL2 approach corresponds to the introduction of new events in an Event-B refinement (see Section 1.2).

An early incarnation of Event-B has already been used in the development of hardware for digital signal processing [5]. A circuit is specified at an abstract level as a recurrence relation, which is refined to derive a pipelined implementation. The pipeline is used specifically to store preceding values for the purpose of implementing the recurrence relation. Other related work is the separate use of classical B [7] and action systems [11] to develop pipelined hardware. However, in both cases, their motivation seems to be somewhat different because they start with specific hardware implementations in mind, which influences the form of their abstract-level specifications. In this paper, the abstract-level specification is derived from the informal description of an instruction set. As a consequence, the subsequent refinement is not as straightforward.

The structure of the paper is as follows. Event-B and its notion of refinement are introduced. Then an informal description of the instruction set of a hypothetical processor (taken from [9]) is presented in Section 2; from this, an abstract Event-B specification is constructed. In Section 2.2, a pipelined hardware architecture (also taken from [9]) is introduced as a candidate processor for the instruction set, and Event-B refinement is used to formalise the architecture and its behaviour with respect to the instruction set. A proof of refinement, described in Section 2.4, fails to highlight some potential hazards in the pipelined architecture, and a novel solution (*event merging*) is proposed in Section 3 as a way to overcome these problems. In Section 4, we see how over-zealous merging can introduce further problems in a refinement.

A conclusion to the work is given in Section 5.

Note that this paper is not questioning the soundness of superposition refinement. The purpose of this paper is, given an abstract specification of an instruction set, to propose a solution to the task of its formal refinement to hardware. Also note, the refined model presented in this paper is still relatively abstract when compared to hardware description languages such as VHDL [6]. Hence, the paper does not propose a complete route to hardware. However, it could be incorporated into any future approach that generates hardware from Event-B specifications.

1.1 The Event-B Notation

An abstract Event-B specification [10] 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 associated 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 that must be discharged to verify that the specification is well-defined and 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, the definition of an event **E** takes the form:

```
EVENT E
    WHEN
        $G(v)$
    THEN
        $S(v)$
    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)[2]. The guard must hold for the substitution to be performed (otherwise the event is *blocked*). A **REFINES** clause, which includes the name of an abstract event, is necessary when the definition is a refinement of an existing event. There are

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

three kinds of generalised substitution in an event: *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.

1.2 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 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 conforming to the behaviour of the abstract specification. A demonstration of Event-B refinement will be given in Section 2.4.

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. However, 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 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 1. If, in a state satisfying the gluing invariant J, a concrete event with (refined) generalised

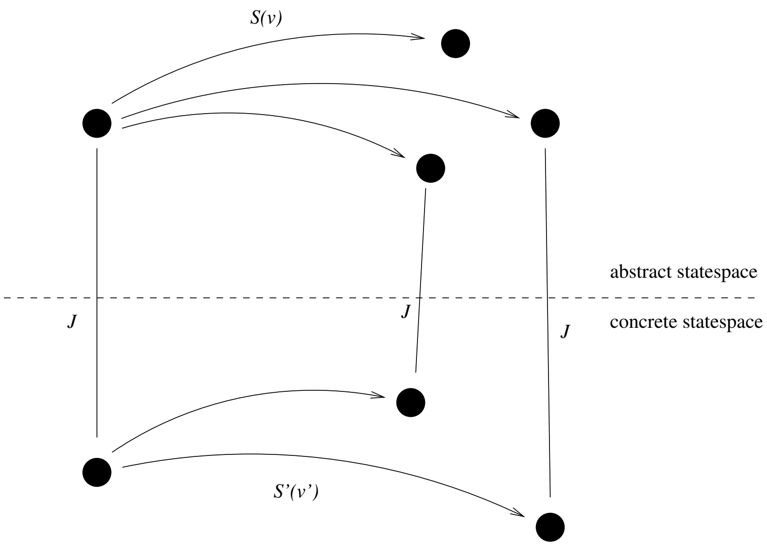

Fig. 1. Refinement of an Existing Event

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 indefinitely, thereby maintaining the visibility of existing events. 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)[3]. Of course, the desired properties of newly introduced events can be incorporated into the invariant, and a proof is required to show that these additional properties are maintained.

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

2 A Simple Example

This example of a pipelined processor is taken from [9] (which itself was taken from [13]). At the most abstract level, we have to consider the instructions accepted by the processor, and their effect on the state of the processor (ignoring how this behaviour is implemented). The state of the processor includes a program counter which records the index of the next instruction to be performed, and a collection of registers that hold values during the execution of a sequence of instructions (i.e. a *program*). We shall consider three instructions: an addition instruction, a conditional branch instruction and an unconditional branch instruction. According to the informal description, each instruction comprises four elements: the opcode (in this case: *add*, *bez* or *jump*), the identity of the target register, and the identities of two source registers. The instructions are defined informally as follows:

- the *add* instruction adds the values held in the two source registers and then stores the result in the target register. Also, the program counter is incremented in order to perform the next instruction;
- the *bez* instruction sets the program counter to the value held in the second source register if the value held in the first source register is 0. If this value is not 0 then the program counter is simply incremented;
- the *jump* instruction unconditionally sets the program counter to the value held in the first source register.

2.1 An Abstract-level Model

At the abstract level (which we call the instruction set architecture (ISA) level), each instruction will be modelled in Event-B as an event which executes instantaneously. This is the most natural way to formalise the instructions listed above because, at this level of abstraction, we are only interested in the effects of the instructions on the state of the processor rather than the method by which the instructions are executed.

A context holds the declaration of all types and constants used in the formal model. This is shown in Figure 2. Note that *Opcode* is an enumerated type consisting of elements *add*, *bez* and *jump*. The set *Reg* contains the identities of the value registers (in this case we only consider two registers *ra* and *rb*). Finally, the set *Instruction* contains all 4-tuples comprising one opcode, the identity of a target register, and the identities of two source registers. The constant functions ins_opcode, ins_target, $ins\_source1$ and $ins\_source2$ are defined to access the components of an instruction (via the projections of the 4-tuples).

```
CONTEXT Types
SETS
    Reg
    Opcode
CONSTANTS
    ra
    rb
    Instruction
    ins_opcode
    ins_target
    ins_source1
    ins_source2
    add
    bez
    jump
AXIOMS
    $Opcode = \{add, bez, jump\}$
    $add \neq bez$
    $add \neq jump$
    $bez \neq jump$
    $Reg = \{ra, rb\}$
    $ra \neq rb$
    $Instruction = Opcode \times Reg \times Reg \times Reg$
    $ins\_opcode \in Instruction \rightarrow Opcode$
    $ins\_target \in Instruction \rightarrow Reg$
    $ins\_source1 \in Instruction \rightarrow Reg$
    $ins\_source2 \in Instruction \rightarrow Reg$
    $\forall i \cdot (i \in Instruction \Rightarrow$
        $ins\_opcode(i) =$
            $prj1(Opcode \times Reg)(prj1(Opcode \times Reg \times Reg)(prj1(Opcode \times Reg \times Reg \times Reg)(i))))$
    $\forall i \cdot (i \in Instruction \Rightarrow$
        $ins\_target(i) =$
            $prj2(Opcode \times Reg)(prj1(Opcode \times Reg \times Reg)(prj1(Opcode \times Reg \times Reg \times Reg)(i))))$
    $\forall i \cdot (i \in Instruction \Rightarrow$
        $ins\_source1(i) = prj2(Opcode \times Reg \times Reg)(prj1(Opcode \times Reg \times Reg \times Reg)(i)))$
    $\forall i \cdot (i \in Instruction \Rightarrow ins\_source2(i) = prj2(Opcode \times Reg \times Reg \times Reg)(i))$
END
```

Fig. 2. The abstract context

The dynamic behaviour of the model is defined in a machine. This is shown in Figure 3. To be fully general, a program is modelled as a variable (called *program*) which is a (partial) function that maps instruction indices to instructions, and the variable PC holds the value of the current instruction index. Hence, when PC is in the domain of *program*, the function application $program(PC)$ returns the next instruction to be executed. The variable *regs* records the entries in all value registers. These variables are initialised to arbitrary values of the appropriate type by using the non-deterministic assignment operator $:\in$.

The guard of an event is enabled only when the instruction indexed by PC contains the appropriate opcode. For example, the event **add_inst** is enabled only when $ins\_opcode(program(PC)) = add$. Note that there are two events corresponding to the *bez* instruction: one event for the zero case (i.e. when the first source register is 0) and one for the non-zero case; the guards determine which of these events is enabled by examining the value held in the first source register.

```
MACHINE ISA

SEES Types
VARIABLES
    $PC$
    $program$
    $regs$
INVARIANTS
    $PC \in \mathbb{N}$
    $program \in \mathbb{N} \rightarrowtail\!\!\!\!\!+ Instruction$
    $regs \in Reg \rightarrow \mathbb{N}$

EVENTS
INITIALISATION
    BEGIN
        $PC :\in \mathbb{N}$
        $program :\in \mathbb{N} \rightarrowtail\!\!\!\!\!+ Instruction$
        $regs :\in Reg \rightarrow \mathbb{N}$
    END

EVENT add_inst
    WHEN
        $PC \in dom(program)$
        $ins\_opcode(program(PC)) = add$
    THEN
        $regs(ins\_target(program(PC))) :=$
            $regs(ins\_source1(program(PC))) + regs(ins\_source2(program(PC)))$
        $PC := PC + 1$
    END

EVENT bez_inst
    WHEN
        $PC \in dom(program)$
        $ins\_opcode(program(PC)) = bez$
        $regs(ins\_source1(program(PC))) = 0$
    THEN
        $PC := regs(ins\_source2(program(PC)))$
    END

EVENT bez_inst2
    WHEN
        $PC \in dom(program)$
        $ins\_opcode(program(PC)) = bez$
        $regs(ins\_source1(program(PC))) \neq 0$
    THEN
        $PC := PC + 1$
    END

EVENT jump_inst
    WHEN
        $PC \in dom(program)$
        $ins\_opcode(program(PC)) = jump$
    THEN
        $PC := regs(ins\_source1(program(PC)))$
    END
END
```

Fig. 3. The abstract machine

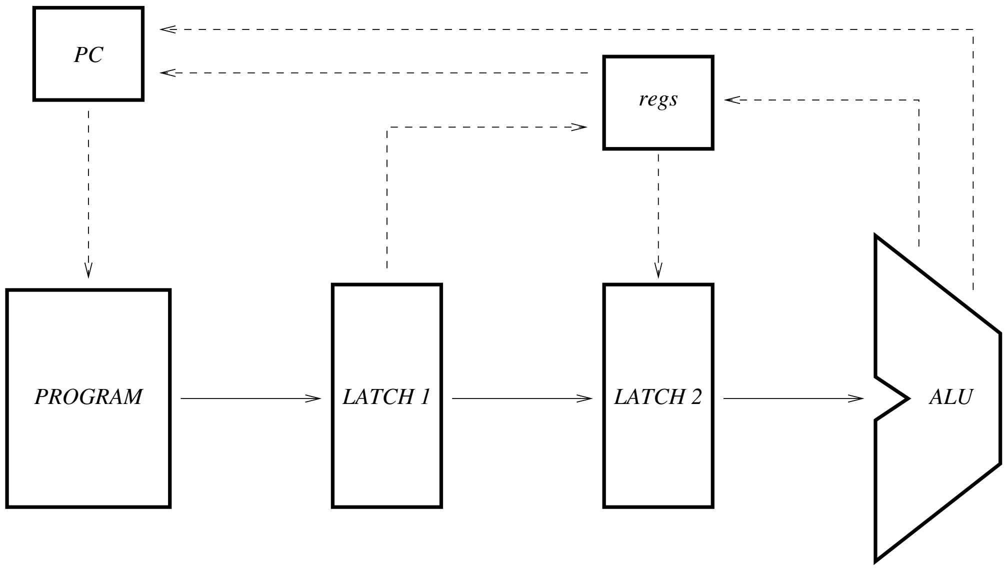

Fig. 4. A three-stage pipeline

2.2 An Implementation

At the implementation level, a pipelined architecture shall be used to execute programs. This is depicted in Figure 4. The solid arrows in the diagram indicate the three stages in the pipeline, whereas the dashed arrows indicate the flow of data to and from the registers and program counter. The first stage of the pipeline fetches the next instruction from memory and puts it in $LATCH1$. The next stage replaces the identities of the source registers (of the instruction held in $LATCH1$) with their actual values. This is stored in $LATCH2$ in readiness for the third stage which performs the necessary ALU operation for the instruction to complete. Note that *add* and *bez* require the arithmetic and logic capabilities of the ALU and, hence, will traverse the entire pipeline. However, the *jump* instruction does not require the ALU and can be completed after reaching $LATCH1$.

Since several instructions can be in the pipeline at any one time, our aim is to prove that the implementation-level architecture faithfully executes the instructions as described at the abstract level. However, even without performing a formal analysis, it is not too difficult to see that, without adequate control mechanisms, it is possible for conflict to arise within the pipeline. For example, consider the following program fragment consisting of two add instructions:

$$add\ ra\ ra\ rb$$
$$add\ rb\ ra\ rb$$

The first instruction adds the values of registers ra and rb and stores the result in ra. The second instruction performs the same addition and stores the result in rb. If both instructions are in the pipeline at the same time then the values of the source registers in the second instruction will be substituted for their register names before the first addition (and subsequent update of ra) has been completed. Hence the addition in the second instruction would be performed on 'old' values. This problem is typically overcome by stalling the second instruction's entry into the pipeline. We therefore expect a formal analysis of the pipeline to expose this hazard.

2.3 An Implementation-level Model

The approach taken in this paper models an implementation-level description as a refinement of the abstract-level description. At the implementation level, the stages of the pipeline will be modelled as distinct events. Hence, to model the execution of a single instruction, an event at the abstract level will be refined to a sequence of events. This is a typical refinement step in Event-B: single atomic events are refined into multiple (i.e. non-atomic) lower-level events. However, as we shall see, in order to model pipelining faithfully, this simple approach is insufficient.

In order to show that the pipeline correctly implements the abstract-level instructions, we construct a refinement of the **ISA** machine and introduce further events to model the stages of the pipeline. Recall from Section 1.2 that refinements of existing events must preserve the behaviour of their abstract counterparts, as dictated by the gluing invariant. In the implementation-level model, we continue to use the variable *regs* and, hence, the gluing invariant is implicitly an identity relation on the values of this variable. Therefore, our definitions of the concrete events **add_inst**, **bez_inst**, **bez_inst2** and **jump_inst** must preserve the behaviour of their abstract counterparts with respect to *regs*. However, as we shall see, these events will be defined in terms of other variables, and this will require us to make assertions about these newly-introduced variables in the implementation-level invariant. Before we introduce the refinement to the **ISA** machine, a refinement of the **Types** context is necessary. This is because the second stage of the pipeline replaces instructions containing the identities of two source registers with instructions containing the *values* of the two source registers. Hence, a new set called *Substitution* is introduced which contains all 4-tuples comprising an opcode, a (target) register identity and two natural numbers, i.e.:

$$Substitution = Opcode \times Reg \times \mathbb{N} \times \mathbb{N}$$

$$\begin{aligned}
&fetchPC \in \mathbb{N}\\
&latch1 \in Instruction\\
&latch2 \in Substitution\\
&latch1\_status \in Status\\
&latch1PC \in \mathbb{N}\\
&latch2\_status \in Status\\
&latch2PC \in \mathbb{N}\\
&commitPC \in \mathbb{N}
\end{aligned}$$

Fig. 5. Implementation-level variables

The functions *sub_opcode*, *sub_target*, *sub_source*1 and *sub_source*2 are defined to access the fields of the 4-tuple. We also introduce a set called *Status* which contains the elements *vacant* and *occupied* to record the status of each latch in the pipeline.

The implementation-level refinement of the **ISA** machine is called **MA** (for *micro-architecture*). The variables introduced at this level, and their types, are shown in Figure 5. The variable *latch*1 models the first stage of the pipeline (i.e. the box labelled $LATCH1$ in Figure 4). If it is occupied (according to the variable *latch*1*_status*) then *latch*1 contains the instruction that has just entered the pipeline. The variable *latch*1*PC* records the index of the instruction held in *latch*1. Similarly, *latch*2 models the second stage of the pipeline (i.e. the box labelled $LATCH2$ in Figure 4). At this stage, the identities of the source registers are replaced by their values. Hence, *latch*2 is of type $Substitution$. Since the pipeline can hold a number of partially completed instructions, we record the index of the next instruction to enter the pipeline ($fetchPC$) and the index of the next instruction to complete (*commitPC*). Recall that the variable PC records the index of the next instruction to complete at the abstract level. Hence, we can make this observation explicit by adding the following equality to the gluing invariant:

$$commitPC = PC$$

As well as the events **add_inst**, **bez_inst**, **bez_inst2** and **jump_inst**, we define two further events at the implementation level which correspond to the first two stages of the pipeline: the event **fetch** loads instructions into *latch*1, and **set_up_alu_op** loads instructions from *latch*1 into *latch*2 (and substitutes the identities of the source registers with their values). These are shown in Figure 6.

```
EVENT fetch
    WHEN
        $fetchPC \in dom(program)$
        $latch1\_status = vacant$
    THEN
        $fetchPC := fetchPC + 1$
        $latch1 := program(fetchPC)$
        $latch1\_status := occupied$
        $latch1PC := fetchPC$
    END

EVENT set_up_alu_op
    WHEN
        $latch1\_status = occupied$
        $latch2\_status = vacant$
        $ins\_opcode(latch1) \in \{bez, add\}$
    THEN
        $latch2 := (ins\_opcode(latch1) \mapsto ins\_target(latch1) \mapsto$
                $regs(ins\_source1(latch1)) \mapsto regs(ins\_source2(latch1)))$
        $latch1\_status := vacant$
        $latch2\_status := occupied$
        $latch2PC := latch1PC$
    END
```

Fig. 6. New events

2.4 A Proof of Refinement

The refined events **add_inst**, **bez_inst**, **bez_inst2** and **jump_inst** are defined in terms of the new latch variables. Examples of these are given in Figure 7. Note that the event **add_inst** uses *latch*2 because it requires the ALU, whereas **jump_inst** does not require the ALU and, hence, can be completed on reaching *latch*1 (but only if *latch*2 is vacant).

On submitting the implementation-level model to the Rodin tool, proof obligations are generated in order to show that **MA** is a refinement of **ISA**. The tool will discharge any proof obligations that it can automatically, leaving the rest for the user. Discharging all proof obligations requires a strengthening of the gluing invariant to capture the implementation-specific relationships between the variables. For completeness, this is shown in Figure 8. Although this invariant appears to be non-trivial, it is quite straightforward to derive this from the outstanding proof obligations.

```
EVENT add_inst
REFINES add_inst
    WHEN
      $sub\_opcode(latch2) = add$
      $latch2\_status = occupied$
    THEN
      $regs(sub\_target(latch2)) := sub\_source1(latch2) + sub\_source2(latch2)$
      $latch2\_status := vacant$
      $commitPC := latch2PC + 1$
    END

EVENT jump_inst
REFINES jump_inst
    WHEN
      $ins\_opcode(latch1) = jump$
      $latch1\_status = occupied$
      $latch2\_status = vacant$
    THEN
      $fetchPC := regs(ins\_source1(latch1))$
      $latch1\_status := vacant$
      $latch2\_status := vacant$
      $commitPC := regs(ins\_source1(latch1))$
    END
```

Fig. 7. Refined events

Once constructed, the invariant can be used to discharge all remaining proof obligations, thereby proving that **MA** is a refinement of **ISA**. This is somewhat disturbing because Section 2.2 described a situation in which conflict could arise within the pipeline. The conflict does not arise in the Event-B model because the events will vacate the latches before they are re-occupied by subsequent instructions. To see why the conflict does not arise, recall the simple example program from Section 2.2:

$$add\ ra\ ra\ rb$$
$$add\ rb\ ra\ rb$$

If this program begins executing in a state in which $ra = 1$ and $rb = 2$ then the resulting final state should be $ra = 3$ and $rb = 5$. The following table shows the steps in the execution using the events defined in **MA**. Note that we begin in a state in which the first *add* instruction has reached *latch*2 and the second *add* instruction has reached *latch*1.

$$
\begin{aligned}
&latch2\_status = occupied \Rightarrow latch2PC = commitPC\\
&latch1\_status = occupied \wedge latch2\_status = vacant \Rightarrow latch1PC = commitPC\\
&latch1\_status = vacant \wedge latch2\_status = vacant \Rightarrow fetchPC = commitPC\\
&latch2\_status = occupied \wedge latch1\_status = vacant \Rightarrow fetchPC = commitPC + 1\\
&latch1\_status = occupied \wedge latch2\_status = vacant \Rightarrow fetchPC = commitPC + 1\\
&latch2\_status = occupied \wedge latch1\_status = occupied \Rightarrow latch1PC = commitPC + 1\\
&latch2\_status = occupied \wedge latch1\_status = occupied \Rightarrow fetchPC = commitPC + 2\\
&latch2\_status = occupied \Rightarrow ins\_opcode(program(PC)) = sub\_opcode(latch2)\\
&latch1\_status = occupied \wedge latch2\_status = vacant \Rightarrow program(PC) = latch1\\
&latch2\_status = occupied \wedge latch1\_status = occupied \Rightarrow program(PC + 1) = latch1\\
&latch2\_status = occupied \wedge sub\_opcode(latch2) = add \Rightarrow\\
&\qquad sub\_target(latch2) = ins\_target(program(PC))\\
&latch2\_status = occupied \Rightarrow\\
&\qquad sub\_source1(latch2) = regs(ins\_source1(program(PC)))\\
&latch2\_status = occupied \Rightarrow\\
&\qquad sub\_source2(latch2) = regs(ins\_source2(program(PC)))
\end{aligned}
$$

Fig. 8. Implementation-level invariant

event	*latch1*	*latch2*	*ra*	*rb*
···	*add rb ra rb*	*add ra* 1 2	1	2
add_inst	*add rb ra rb*	—	3	2
set_up_alu_op	—	*add rb* 3 2	3	2
add_inst	—	—	3	5

In this initial state, the only enabled event is **add_inst**, which takes the (substituted) instruction in *latch*2, performs the addition and updates the registers. Only after this has happened and *latch*2 has been vacated can the **set_up_alu_op** event perform the substitution to move the instruction in *latch*1 to *latch*2.

The conflict only arises because an (unconstrained) pipeline would perform the two steps at once, and we would observe the following behaviour:

event	$latch1$	$latch2$	ra	rb
$\cdots$	$add\ rb\ ra\ rb$	$add\ ra\ 1\ 2$	1	2
add_inst/set_up_alu_op	—	$add\ rb\ 1\ 2$	3	2
add_inst	—	—	3	3

Here we can observe an incorrect final state.

3 Merging Events

In order to expose this conflict in the formal model, it is possible to define extra events. In addition to the events defined earlier, we define events that merge the stages of the pipeline. Examples of merging are shown in Figure 9. The combined event **set_up_alu_op_and_fetch** uses the events shown in Figure 6. Note that we have not simply combined the guards and actions of the two events because it is possible to eliminate some redundant features. For example, in this definition we have not included the guard $latch1\_status = vacant$ from **fetch** because one of the actions of **set_up_alu_op** vacates $latch1$. Similarly, we can eliminate the action $latch1\_status := occupied$ from **fetch** because the guard of **set_up_alu_op** assumes this to be true. The combined event **add_inst_and_fetch** illustrates the combination of newly introduced events (in this case **fetch**) with refined events (**add_inst**). Hence, this combined event is defined to be a refinement of the abstract **add_inst** event.

Conflict is exposed when we cannot discharge all proof obligations generated by the merged events. Note that during this process we keep the invariant fixed: merged events must continue to maintain the same invariant. This eliminates an additional level of complexity in the merging process. Conflict, in this case, arises when we try to combine **add_inst** with **set_up_alu_op**. By following the same approach, the combination **add_inst_and_set_up_alu_op** is shown in Figure 10. Since this is defined to be a refinement of the abstract **add_inst** event, we are required to prove that its behaviour is preserved (as depicted in Figure 1). Within the invariant shown in Figure 8 are the following two clauses:

$$\begin{aligned} &latch2\_status = occupied \ \wedge \ sub\_opcode(latch2) = add \ \Rightarrow \\ &\qquad sub\_source1(latch2) = regs(ins\_source1(program(PC))) \end{aligned}$$

```
EVENT set_up_alu_op_and_fetch
    WHEN
        $latch1\_status = occupied$
        $latch2\_status = vacant$
        $fetchPC \in dom(program)$
        $ins\_opcode(latch1) \in \{bez, add\}$
    THEN
        $latch1 := program(fetchPC)$
        $latch2 := (ins\_opcode(latch1) \mapsto ins\_target(latch1) \mapsto$
                  $regs(ins\_source1(latch1)) \mapsto regs(ins\_source2(latch1)))$
        $fetchPC := fetchPC + 1$
        $latch2\_status := occupied$
        $latch1PC := fetchPC$
        $latch2PC := latch1PC$
    END


EVENT add_inst_and_fetch
REFINES add_inst
    WHEN
        $sub\_opcode(latch2) = add$
        $latch2\_status = occupied$
        $latch1\_status = vacant$
        $fetchPC \in dom(program)$
    THEN
        $regs(sub\_target(latch2)) := sub\_source1(latch2)+sub\_source2(latch2)$
        $latch2\_status := vacant$
        $commitPC := latch2PC + 1$
        $fetchPC := fetchPC + 1$
        $latch1 := program(fetchPC)$
        $latch1\_status := occupied$
        $latch1PC := fetchPC$
    END
```

Fig. 9. Merged events

$$\begin{aligned}&latch2\_status = occupied \;\wedge\; sub\_opcode(latch2) = add \;\Rightarrow\\ &\qquad sub\_source2(latch2) = regs(ins\_source2(program(PC)))\end{aligned}$$

These state that the operands of the add instruction in the implementation and

```
EVENT add_inst_and_set_up_alu_op
REFINES add_inst
    WHEN
      $sub\_opcode(latch2) = add$
      $latch2\_status = occupied$
      $latch1\_status = occupied$
      $ins\_opcode(latch1) \in \{bez, add\}$
    THEN
      $regs(sub\_target(latch2)) := sub\_source1(latch2) + sub\_source2(latch2)$
      $commitPC := latch2PC + 1$
      $latch2 := (ins\_opcode(latch1) \mapsto ins\_target(latch1) \mapsto$
                  $regs(ins\_source1(latch1)) \mapsto regs(ins\_source2(latch1)))$
      $latch2PC := latch1PC$
      $latch1\_status := vacant$
    END
```

Fig. 10. A conflicting combination

abstract models match. By introducing the event **add_inst_and_set_up_alu_op**, these clauses yield two proof obligations which, after simplification, require proofs to the following:

$$regs(ins\_source1(latch1)) = (regs \mathbin{\lhd\!\!-} \{sub\_target(latch2) \mapsto \cdots\})(ins\_source1(latch1))$$

$$regs(ins\_source2(latch1)) = (regs \mathbin{\lhd\!\!-} \{sub\_target(latch2) \mapsto \cdots\})(ins\_source2(latch1))$$

where $\lhd\!\!-$ overrides a function with new values. These two proof obligations are, in fact, asking us to prove that there is no conflict between $latch1$ and $latch2$ because the equality of the function applications can be proven only if $sub\_target(latch2) \neq ins\_source1(latch1)$ and $sub\_target(latch2) \neq ins\_source2(latch1)$. Indeed, by adding these inequalities to the guard of **add_inst_and_set_up_alu_op** the proof obligations are discharged automatically. As a consequence, by strengthening the guard in this way, the event will not be enabled in a conflicting situation, and the previously-defined events **add_inst** and **set_up_alu_op** will have to be called separately so that the conflicting instructions make progress along the pipeline. This has revealed the stalling mechanism that is necessary to delay the second add instruction until the first one has completed.

The merging step presented in this section is very similar to the synchronised parallel composition of event systems in [4], in which B machines (representing system components) are combined into a single machine: events are merged and guards are strengthened according to an accompanying *synchronisation specification*. It would be possible to achieve some of the results described in this section using the approach presented in [4], and it would be advantageous to have a synchronised specification to declare which events should be merged. In particular, it would be possible to combine a machine with itself in order to merge events contained in the same machine. However, the pipeline example demonstrates how merging through refinement can be a useful way to determine which guards need to be strengthened (as illustrated in the refinement of **add_inst** by **add_inst_and_set_up_alu_op**). The main result in [4] shows that their synchronisation step is monotonic with respect to refinement, but does not consider synchronisation as a refinement step itself. As far as the author is aware, there are currently no plans to implement the results of [4] in the Rodin tool.

4 A Step too far

In [9], the refinement is capable of performing an *add* instruction and a *jump* instruction in the same step because (as stated in Section 2.2) the *jump* instruction can be performed when it reaches *latch*1. If we attempt to do this using the merging technique then we get the definition shown in Figure 11. The first question that arises is which of the abstract operations does it refine? It does not refine **add_inst** because, among other things, this abstract event increments the program counter *PC* (see Figure 3). Similarly, it does not refine **jump_inst** because, unlike **add_inst_and_jump_inst**, this abstract event does not change the values held in the registers. However, whether Event-B is used or any other approach, it is difficult to see how such a combination can be considered to be a refinement at all, because **add_inst** and **jump_inst** exist as distinct observable events at the abstract level, and the merged event does not preserve this level of granularity. It is, therefore, unclear how this is permitted in the ACL2 example of [9] since this anomaly is not specific to the Event-B model[4].

5 Conclusion

This paper has demonstrated an approach to the formal development of pipelined hardware in Event-B. It has shown that it is possible to overlook potential hazards if Event-B's notion of refinement is used naïvely. The approach presented

[4] An answer to this question was requested but not received.

```
EVENT add_inst_and_jump_inst
REFINES ?
    WHEN
        $sub\_opcode(latch2) = add$
        $latch2\_status = occupied$
        $latch1\_status = occupied$
        $ins\_opcode(latch1) = jump$
        $ins\_source1(latch1) \neq sub\_target(latch2)$
    THEN
        $regs(sub\_target(latch2)) := sub\_source1(latch2) + sub\_source2(latch2)$
        $latch2\_status := vacant$
        $fetchPC := regs(ins\_source1(latch1))$
        $commitPC := regs(ins\_source1(latch1))$
        $latch1\_status := vacant$
    END
END
```

Fig. 11. A bad combination

in this paper of merging events gives a more accurate model of pipelined behaviour, and reveals conflicts within the pipeline by generating unprovable proof obligations. By strengthening guards, it is possible to circumvent such conflicts, and introduces explicit constraints in the pipeline to guarantee correct execution. An alternative solution could be achieved by redefining the abstract-level specification, but this would obscure the relationship between the formal specification and the informal description of the instruction set.

Even though Event-B's superposition refinement did not expose all of the possible problems of the pipeline example, there are definite advantages to modelling and refining using this technique: in general, it enables the introduction of specific parts of a system in a stepwise manner without having to consider the global view of the system. However, since we have demonstrated that is possible to overlook potential hazards, we must conclude that awareness of 'global' behaviour is also necessary when using this approach.

The merge step presented in Section 3 introduces another level of complexity to formal modelling: for a given set of events, the number of possible merge events is considerable. However, not all combinations are 'meaningful'. For example, whilst it is meaningful to have events such as **set_up_alu_op_and_fetch**, which moves instructions from *latch*1 to *latch*2 and then loads *latch*1 in a single step, it does not make sense to have a **fetch_and_set_up_alu_op** event because this would load an instruction into *latch*1 and immediately move it into *latch*2.

Keeping the invariant fixed during the merging step is another way in which complexity is reduced. It should not be necessary to modify the invariant during merging because the merged events do not add anything new — they merely allow several existing events to occur in a single step. Hence, the invariant should continue to capture the properties of the system. Any unprovable proof obligations will be a direct result of conflict within a merged event and, hence, reveal a hazard on the pipeline.

Acknowledgement

The author is grateful to the reviewers for their valuable comments.

References

[1] Abrial J. R.: *The B Book: Assigning Programs to Meaning*, CUP (1996).

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

[3] Back R. J., Sere K.: *Superposition Refinement of Reactive Systems*. In Formal Aspects of Computing, Springer (1996).

[4] Bellegarde F., Julliand J., Kouchnarenko O.: *Synchronised Parallel Composition of Event Systems in B*. In ZB 2002: Formal Specification and Development in Z and B, Springer (2002).

[5] Hallerstede S., Zimmermann Y.: *Circuit Design by Refinement in EventB*. In FDL'04, Forum on Specification and Design Languages (2004).

[6] *IEEE Standard VHDL Language Reference Manual*, IEEE Computer Society (2000).

[7] Ifill W.: *The Formal Development of an Example Processor in AMN, C and VHDL*, MSc thesis, Royal Holloway, University of London (1999).

[8] Linderholm T., Yellin F.: *The Java^{TM} Virtual Machine Specification, Second Edition*, Addison-Wesley (1999).

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

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

[11] Plosila J., Sere K.: *Action Systems in Pipelined Processor Design*. Proc. of the International Symposium on Advanced Research in Asynchronous Circuits and Systems, IEEE Computer Science Press (1997).

[12] Namjoshi K. S.: *A Simple Characterisation of Stuttering Bisimulation*. Proc. of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science, LNCS (1997).

[13] Sawada J.: *Verification of a Simple Pipelined Machine Model*. In M. Kaufmann, P. Manolios and J. S. Moore (editors) *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers (2000).

[14] Voisin L.: *User Manual of the Rodin Platform*, version 2.2 (2007).