Using Event-B to construct instruction set architectures

Stephen Wright, Kerstin Eder

Using Event-B to Construct Instruction Set Architectures

Stephen Wright[1] and Kerstin Eder[1]

[1]Department of Computer Science, University of Bristol, UK

Abstract. The Instruction Set Architecture (ISA) of a computing machine is the definition of the binary instructions, registers, and memory space visible to an executable binary image. ISAs are typically implemented in hardware as microprocessors, but also in software running on a host processor, i.e. Virtual Machines (VMs). Despite there being many ISAs in existence, all share a set of core properties which have been tailored to their particular applications. An abstract model may capture these generic properties and be subsequently refined to a particular machine, providing a reusable template for development of robust ISAs by the formal construction of all normal and exception conditions for each instruction. This is a task to which the Event-B [MAV05, Sch01] formal notation is well suited. This paper describes a project to use the Rodin tool-set [ABH06] to perform such a process, ultimately producing two variants of the MIDAS (Microprocessor Instruction and Data Abstraction System) ISA [Wri08,Wri09/1] as VMs. The abstract model is incrementally refined to variant models capable of automatic translation to C source code, which this is compiled to create useable VMs. These are capable of running binary executables compiled from high-level languages such as C [KR88], and compilers targeted to each variant allow demonstration programs to be executed on them.

1. Introduction

One of the most fascinating aspects of a computing machine is its instructions: a relatively small set of obscure functions that, when combined in large sequences, forms the emergent properties of a computer program. Thus, the same machine may be used to calculate a tax return, play a video game, or keep an otherwise uncontrollable aircraft in the air, only by changing the seemingly unfathomable binary sequence loaded into its memory. However, the lack of intuitive connection between the detail of a computing machine‘s instructions and the large-scale functionality of a program presents a problem in attempting to define those instructions: how to ensure the machine reliably handles all combinations of instructions, whether functionally correct or erroneous. This problem has only been exacerbated with the appearance of Reduced Instruction Set Computer machines, in which the instruction set is designed purely for efficiency and simplicity, with no regard to easing the burden of a programmer attempting to write programs or understand the rationale of an instruction sequence. This has motivated a method for formally managing this comprehension gap through the use of abstraction (the simplification of a system description to an understandable level), and subsequent refinement (the incremental additional of detail in understandable steps).

2. Related Work

Since the 1980s there has been much activity in the field of automatic verification of processor microarchitecture (i.e. the hardware used to implement the ISA). Two approaches have been taken:

definition of simplified academic examples capable of being studied in their entirety [GB90,Hun94], and verification of fragments of pre-existing commercial processors [BH91,Fox03,SM95]. A common pattern emerges throughout these projects. Instruction sets are defined as a series of individual specifications defining the behavior of some or all of the entire instruction set. These target-specific specifications are then used to manually or semi-automatically prove theorems stating the assumed adherence to these behavioural descriptions by the microarchitecture, usually down to the level of gate logic.

In the domain of VMs, the ubiquitous Java Virtual Machine (JVM) has attracted much interest in its formal verification. JVM implementations include an instruction interpreter (which performs the functions of a microprocessor's ISA) as well as other components. Architectural models have been constructed [SSB01], and other components formally specified and verified, such as the byte-code verifier [Cas02,KN01] and parts of the instruction interpreter [Qia99]. At implementation level, Event-B has been used to construct a specification of some of the JVM instruction set in order verify an existing micro-coded interpreter implementation [EG07].

Such work highlights the need for systematic construction of the specification itself: instruction sets have been defined as a series of individual specifications for each instruction, making any verification effort vulnerable to errors at this point. Such errors are generally only discovered by manual review or mismatch with the very implementations they are intended to check. In [BH91] "a few" errors in the specification are reported and two discussed in detail. One is a simple type mismatch, but another is more serious in which handling of a possible invalid instruction during instruction fetching is omitted. In [SM95] twenty-eight bugs in a formal specification were found by manual review, in comparison with the two bugs finally found in the implementation

3. Event-B and Rodin

Event-B [MAV05] is a formal method that combines mathematical techniques, set theory, and first-order logic. It is an evolution of B-Method [Abr96] (now often referred to as Classic-B), which was itself derived from Z notation [Spi89]. Event-B is a simplification of Classic-B, decomposing machines into small, discrete events explicitly linked to their abstractions. This allows more convenient practical development by encouraging more incremental refinement and allowing easier verification of logical proofs. Event-B's other advantage is its flexibility, both in the notation itself and its supporting tools. For example, Event-B does not strictly define a notation, but a methodology for the construction and analysis of linked logical objects, and the notation seen when using support tools is actually an arbitrary front-end for users familiar with Classic-B and Z notation. Event-B was therefore developed closely alongside a supporting tool-set, Rodin [ABH06], which allows flexibility in the presentation and manipulation of an underlying linked database representing a formal model. Rodin has a modular architecture based on the Eclipse framework [Ecl09] with clearly defined Application Programming Interfaces exposing the database and supporting expansion via the addition of Eclipse plug-ins.

4. Common Properties of Instruction Set Architectures

In spite of a vast number of general-purpose microprocessor ISAs in existence, all share a set of core properties [HP03]. Programs are stored within the machine as a contiguous array of binary values, each locatable via an integer address stored in a special register: the program counter (PC). The selection of the next instruction to be executed ("control flow") is achieved by simple incrementing of the PC, or its overwriting with a calculated value to perform a jump operation. Taking one or two elements of a bank of fixed-size registers (the "register file"), and outputting a result back to it performs calculations. In the case of a destructive operation, the output register may be one of the input registers. Certain ISAs implement specialist instructions with more than two inputs [AMD07], but are not targeted at general purpose applications. Calculations consist of bit-manipulations, typically implementing the basic arithmetic operations (i.e. addition, subtraction, multiplication and division) plus a number of application-oriented operations. Data may be loaded into the register file via “immediate mode” addressing, in which a constant value is incorporated in the executed instruction, moved between registers via “register direct mode” addressing, or transferred from a specified location in a read-writable data memory via “register indirect mode”. Register indirect mode may also be used to copy data back to this memory from the register file. For all addressing modes source and destination registers are specified by index values incorporated within the instruction. Variations on this general paradigm exist, such as PC-relative mode [Hit98], which may be considered a form of indirect addressing. The data memory consists of a contiguous array of binary values: large register values may be stored across two or more contiguous memory locations.

In order to execute binary images compiled from the C high level language using existing tools, particularly the GNU Compiler Collection (GCC), the ISA must be further specified to implement multiple thirty-two bit general registers, eight bit wide data memory and all instruction and data memory addressing from a single register [Sta01].

5. The MIDAS VM

In order to demonstrate the completeness and usefulness of the modeling technique, a working ISA was developed. Implementation of an existing instruction set and machine architecture was considered: such an approach would allow use of existing support tools and benefit from existing development. Two example machines were considered: an existing Virtual Machine standard, the Java Virtual Machine (JVM), and a typical hardware microprocessor targeted at embedded systems, the Hitachi SH4. Existing architectures and instruction sets contain complexities related to achieving particular requirements not relevant to this application. For example, the JVM contains various features to allow efficient support for the Java high level language [LY99], and the SH4 instruction set’s use of delay-slotted branch instructions [Hit98], enhances performance in a hardware implementation but increases processor and support tool complexity. Full control over the design allows the instruction set to be reduced to a minimum required for program execution and instruction code formats to be selected that are logically based on the formal model.

The MIDAS (Microprocessor Instruction & Data Abstraction System) specification [Wri09/1] describes two variants of a Modified Harvard Architecture, thirty two bit register ISA with a total of thirty-five instructions in eight orthogonal groups [HP03], and a little-endian memory system. The two variants are a stack-based machine and randomly accessible register array machine [HP03], employing the same numerical values to implement similar instruction functionality, the differences being limited to Register File behavior. Randomly accessible register array machines are now the most common form of hardware-implemented microprocessors [HP03], but stack machines are still used, particularly in the field of true VMs not intended for hardware implementation [LY99]. Therefore, demonstration of both architectures is desirable.

The MIDAS instruction groups implement no-operation (one instruction), register-fetch (seven instructions), register-store (four instructions), single-operand calculations (two instructions), dual-operand calculations (thirteen instructions), operand compare (five instructions), control-flow jump (two instructions), and machine halt (one instruction). A Harvard Architecture was selected for its guaranteed prevention of executable corruption by bad data accesses, increasing integrity in safety-critical applications. In order to reduce ISA size and complexity the MIDAS ISA is not optimized for performance: for example the single operand compare-to-zero instruction provided by many typical ISAs is not implemented [Hit98]. The basic instruction is an eight bit field, treated as two four bit sub-fields (nibbles). The instruction group is specified by the most significant nibble (MSN) and the precise operation given by a modifier in the least significant nibble (LSN). Instruction groups and modifiers are derived from the groupings constructed in the formal model, allowing for an efficient decoding scheme to be implemented, as events applying to whole instruction groups need only decode the MSN.

6. Model Description

6.1 High-Level Structure

The Event-B model addresses only those aspects of a microprocessor's functionality needed to specify an ISA. Therefore, only a functional description of the instruction set and its associated aspects are included, such as those parts of the memory system visible to the instruction stream. No consideration of the mechanisms within a typical microprocessor needed to efficiently implement an ISA is necessary. For example the model does not address instruction pipelining, memory caching or speculative execution mechanisms [HP03]. The model consists of a common refinement chain, which is then split into separate refinements for two variants of an example ISA constructed for demonstration of the technique. The refinement structure is summarized in Figure 1.

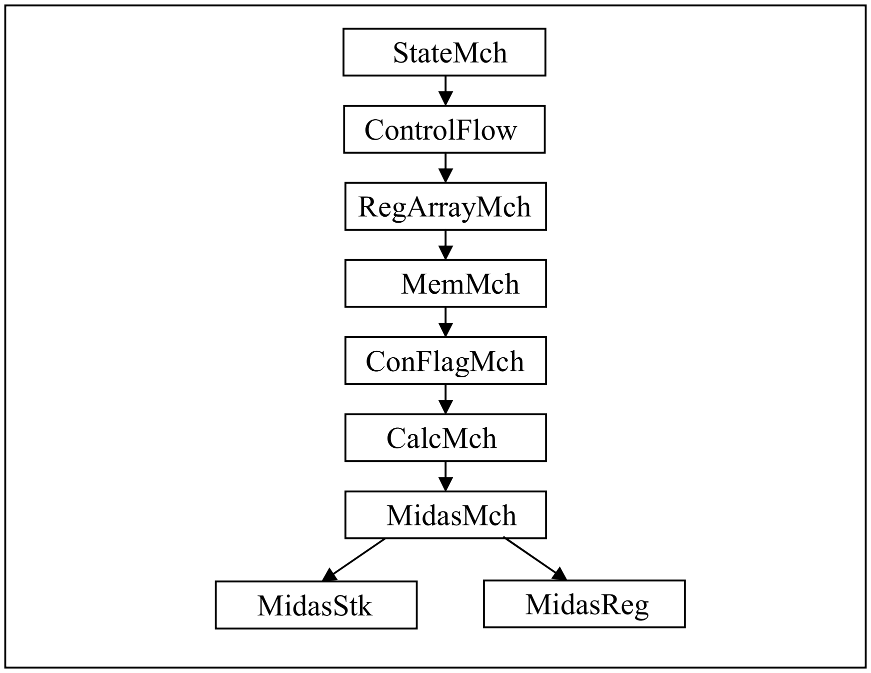

Figure 1: Model Refinement Structure

The model consists of twenty-eight stages of common refinement, which may be summarized into seven distinct layers: StateMch, ControlFlow, RegArrayMch, MemMch, ConFlagMch, CalcMch and MidasMch. Each layer confers particular properties on the model, described individually. The MidasStk refinement branch introduces a further five layers of refinement: the MidasReg branch another two.
Consideration is given to the order of model refinement in order to allow the model to be re-used with minimal modifications. Some layers of the model provide essential definitions necessary to facilitate later refinements, and therefore these must be positioned earlier in the refinement process. For example, the

StateMch layer is required to define the exception condition concept and its triggering of machine deadlock, before any specific exception conditions, such as program counter out-of-range detection, may be defined in the ControlFlow. Other refinements may be performed at any point in the modeling process, and are therefore postponed to as late as possible in order to maximize the flexibility of the model. For example, the refinements specifying the exact calculations supported by the MIDAS VM in CalcMch are positioned prior to only the final MidasMch layer. The decision to place the CalcMch layer after the ConFlagMch layer is made on the assumption that supported calculation operations will be modified more regularly than the mechanism used to implement conditionality. Refinements describing the higher level architectural features of the machine, such as the register and memory address spaces, may be easily identified and placed before implementation-specific features, such as exact instruction code values. Decisions on the relative importance of separate architectural issues are harder to make, and judgment based on projected applications is required. However, such an approach does not consider the consequences of a large number of events being constructed early in the refinement process, and therefore propagating the associated management and proving burden to all subsequent refinements.

The separate modeling of executable and data memories allows its applicability to Harvard or Von Neumann memory architectures (i.e. whether the loaded binary program is visible or write-able in the machine's memory system) [LBSL97,Lee89]. In the case of the MIDAS demonstrator, a Modified Harvard Architecture, in which the read-only region is visible but the binary program is not, is selected at the MidasMch layer.

6.2 The StateMch Layer

StateMch initially abstracts the ISA to the most trivial possible form by defining a single unguarded event acting on only two state variables: an instruction and a machine status, shown in the simplified event in Equation (1).

```
Iterate ≙                                  (1)
BEGIN
  act1: inst :∈ INST
  act2: status :∈ STATE
END
```

The instruction is a member of *INST*, defined as an abstract SET in the accompanying context, which represents the complete instruction space of the machine (i.e. both valid and invalid instructions). Thus all aspects of the actual instruction, including its symbolic representation and method of extraction, are abstracted. The StateMch layer then uses four more refinement steps to construct the state machine illustrated in

Figure 2, defining the modes of operation of the machine and their possible transitions.

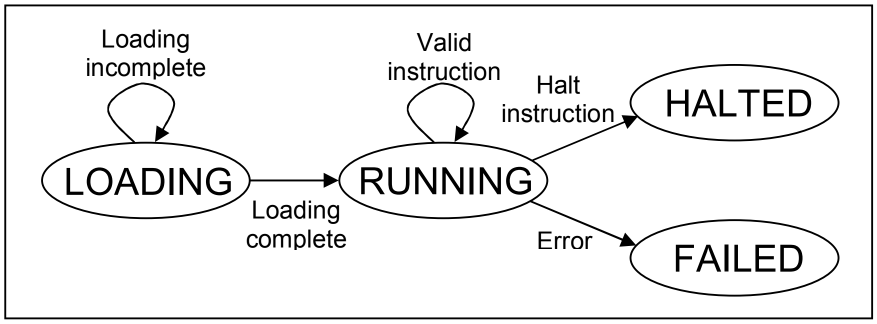

Figure 2: StateMch State Transitions

These modes are LOADING (of the program to be executed), RUNNING (of the loaded program), HALTED (i.e. commanded shutdown of the machine) and FAILED (i.e. detection of an error in the program). The model is initialized to LOADING and may continue in that state until loading is complete, when RUNNING is entered. From RUNNING, RUNNING is re-entered on successful execution of a valid instruction, HALTED is entered by the execution of a valid halt instruction, or FAILED entered by execution of an invalid instruction or unsuccessful execution of a valid instruction. Execution results in the modification of *inst* by unspecified means. Once HALTED or FAILED is entered, the machine enters explicit deadlock by the definition of guarded events with no actions defined. Thus a mechanism for the refining of execution exceptions and the guaranteed stopping of the machine in this event is defined.

Successive division of the INST SET defines instruction groups affecting the state machine. Statements are introduced to explicitly state that the derived sub-sets are inclusive of the entire parent set (in order to ensure that the entire instruction space is decoded) and that the constructed sub-sets are mutually exclusive of each other (to ensure that instruction sub-sets take only one set of properties). This technique is used throughout the model to hierarchically derive all instruction sub-sets. For example, the Event-B fragment shown in Equation (2) describes the initial decomposition of *INST* into the valid and invalid instructions within the instruction space. In the example the *ValidInst* sub-set is explicitly stated as non-empty, whilst the *InvalidInst* sub-set is not. This represents the possibility of a machine in which all possible instruction values are populated with decoded instructions, but a machine with no instructions cannot occur. The actual numerical values of the instruction sub-sets is a feature of a particular ISA, and is therefore postponed to the last refinement stages of the model.

```
CONSTANTS
  ValidInst   // Valid instructions
  InvalidInst // Invalid instructions
AXIOMS
  ValidInst ⊆ INST    // Subset of all instructions
  ValidInst ≠ ∅        // Some valid instructions must exist
  InvalidInst ⊆ INST  // Subset of all instructions
  ValidInst ∩ InvalidInst = ∅     // Mutually exclusive
  ValidInst ∪ InvalidInst = INST // Complete coverage
```
(**2**)

An example refinement is shown in Equation (3), in which the detection of an invalid instruction leading the FAILED state being entered is constructed.

BadInst ≜
WHEN
  grd1: inst ∈ InvalidInst
  grd2: status ∈ RUNNING
THEN
  act1: status := FAILED
END
	(3)

6.3 The ControlFlow Layer

In StateMch, the selection of the instruction is abstracted by the apparent modification of a single instruction variable *inst*. ControlFlow refines this abstraction to a mechanism common to most computing machines: selection of the instruction from a fixed array *instArray* via a modifiable indexing variable *instPtr*, representing the PC. The invariants establishing this refinement are given in Equation (4).

INVARIANTS
  typedef1: instArray ∈ InstArrayDom → INST
  typedef2: instPtr ∈ InstArrayDom
  gluing: inst = instArray(instPtr)
	(4)

ControlFlow uses four more refinement steps to construct the mechanisms by which the PC may be modified (shown in Figure 3), or its attempted modification to illegal values detected. Two instruction groups are defined by their action on the PC: *IncrInst* which increments the PC by a fixed size, and *FlowInst* which allows it to be overwritten by a computed value (i.e. a jump or branch). Events guarded by *FlowInst* are further refined to introduce conditionality and separate PC-relative or absolute calculation of the overwriting vector.

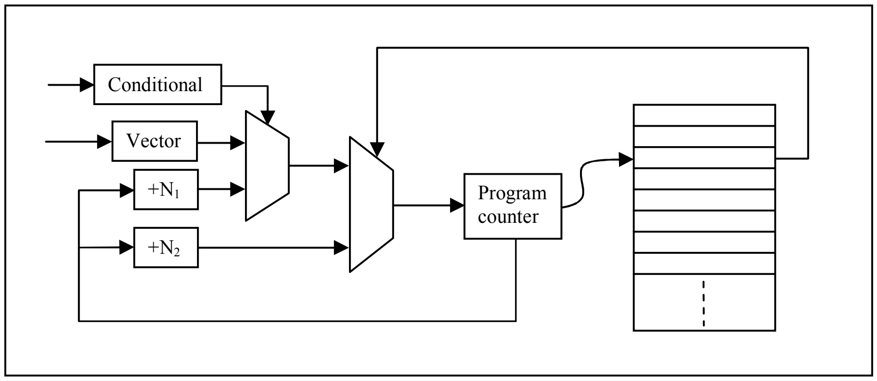

Figure 3: ControlFlow Actions

6.4 The RegArrayMch Layer

The RegArrayMch layer introduces a state variable *regArray*, representing the machine's register file. The registers are defined as an indexed array of indeterminate type *DATA* by the invariant shown in Equation (5).

$regArray \in RegArrayDom \rightarrow DATA$	(5)

Definition of the registers as an array allows the greatest flexibility for the construction of different register architectures in later refinements. Conventional randomly accessible arrays may be constructed by defining indexes free to select any value in the domain *RegArrayDom*. Fixed, specialist registers may also be constructed by the selection of an element via a fixed index and subsequently refined to a separate data element if desired. Stack architectures may be constructed by the refinement of the selecting index to a global stack-pointer variable.

An instruction sub-set allowing the modification of this newly introduced state is initially constructed by dividing the normal, PC-incrementing instruction using the previously described method, creating *NullInst* and *RegWriteInst* sub-sets. This method of division of *NullInst* is used throughout the model to construct instructions capable of modifying newly introduced state variables. Further refinement steps within RegArrayMch sub-divide *RegWriteInst* into different operation types, summarized in Figure 4. *FetchInst* imports data from outside the register file. *MoveInst* transfers data between locations in the register file unchanged. *SingleOpInst* populates a member of the register file with the result of a function that takes a single member of the file as input. *DualOpInst* populates a member of the register file with the result of a function that takes two members of the file as input.

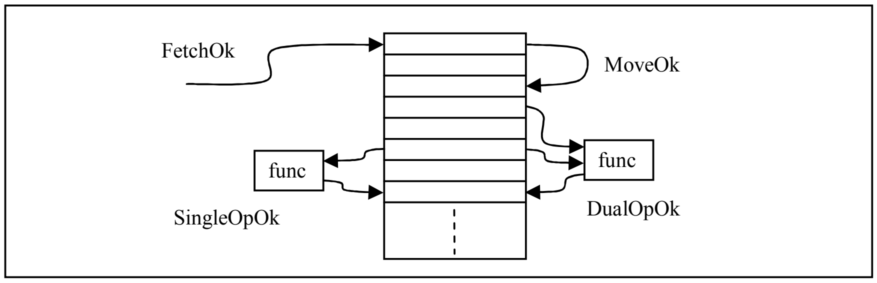

Figure 4: RegArrayMch Data Movements

FetchInst is further refined to specify three possible external sources of data: the numerical value of the PC represented as data, an item of immediate data (i.e. data embedded within the instruction itself), and data from the as yet undefined memory system.

6.5 The MemMch Layer

MemMch introduces another new state variable *memArray*, representing the machine's memory system. The memory is again defined as an indexed array of indeterminate type *DATA* by the invariant shown in Equation (6).

$memArray \in MemArrayDom \rightarrow DATA$	(6)

Division of *NullInst* is again used to construct a modifying instruction *MemWriteInst*. Unlike the multiple refinements anticipated for the *regArray* state, refinement of *memArray* to only a single paradigm is expected: a randomly accessible data array, indexed from a selected element of the register file (see Figure 5).

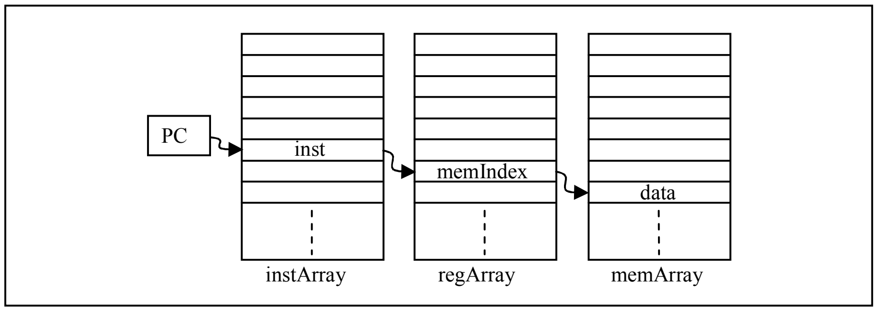

Figure 5: MemMch location selection

This reflects the universality of this pattern in most computing machines [HP03] and is incrementally constructed by the first five refinement steps of MemMch.

The final four refinement steps construct a function performed by most computing machines: the storage of thirty-two and sixteen bit data structures across multiple locations of an eight bit memory system [HP03]. The details of this refinement are given in Section 7.4.

6.6 The ConFlagMch Layer

ConFlagMch introduces a new boolean state variable *conFlag* to refine the abstract conditional defined in the jump instruction constructed in the ControlFlow layer. Use of a separately modified boolean flag is representative of some ISAs [Hit98] but not all [Pat07]. Therefore, this may be regarded as the point at which the refinement process begins to specify a particular ISA from the much more general model constructed in the previous layers.

Division of *NullInst* is again used to construct an abstract instruction capable of modifying the flag, which is then split into instructions performing comparisons of two register file elements: equal, integer-greater-than, integer-less-than, float-greater-than and float -less-than.

6.7 The CalcMch Layer

CalcMch continues the specification of an exact ISA by the splitting of *SingleOpInst* and *DualOpInst* defined in RegArrayMch into the actual functions supported by the ISA. *SingleOpInst* is refined to integer-to-floating-point cast and floating-point-to-integer cast. *DualOpInst* is refined to addition, subtraction, multiplication and division for both integer and floating point, bit-wise OR, AND, XOR, bitmap shift-left and shift-right.

6.8 The MidasMch Layer

MidasMch specifies the remaining details particular to the MIDAS ISA, excepting the register file differences between the stack and register variants. Jump vector sources are specified as register elements. Instruction sizes are fixed at constants (although still undefined). Instructions not implemented by MIDAS (i.e. sixteen bit fetch-immediate) are eliminated. Precise numerical values are assigned to each instruction via a mapping function between the *INST* SET and a range of the natural numbers, shown in Equation (7).

```
CONSTANTS
  Int2Inst   // Mapping function
AXIOMS
  Int2Inst ∈ 0..255 ⤖ INST
  Inst2Inst(34) ∈ FetchImmByteInst
  Inst2Inst(35) ∈ FetchImmLongInst
THEOREMS
  (x≥34) ∧ (x≤35) ⇒ Inst2Inst(x) ∈ FetchImmInst
```
(7)

The separate arrays *instArray* and *memArray* are merged into a single data array *memByte* using the gluing invariants shown in Equation (8).

```
INVARIANTS
memByte ∈ MemDom → INST
∀x.x∈InstArrayDom ⇒
 instArray(x)=(DataByte2Int;Int2Inst)(memByte(x))
∀x.x∈MemArrayDom ⇒ memArray(x)=memByte(x)
```
(8)

Thus concrete numerical values may be allocated to the domains of *instArray* and *memArray* within the single domain of *memByte,* defining the memory map of the machine.

The event associated with the LOADING mode is refined to a state machine capable of sequentially loading the instruction and read-only regions of the new contiguous memory space. Memory-write events are refined to define a small region of writable IO within the memory region.

6.9 The MidasStkMch Layer

MidasStkMch introduces all refinements specific to the stack variant of the MIDAS register file. The register specifier index for direct mode addressing is initially refined to an offset relative to a datum index, which is subsequently refined to the top value of the stack. A new variable *stkSize* is introduced and register access events are refined to access *regArray* relative to it.

The *stkSize* variable is type defined by the INVARIANT shown in Equation (9), thus enforcing the introduction of guards defending against stack overflow and underflow, and the corresponding error events if these conditions are entered.

```
stkSize ∈ 0..MaxRegArraySize
```
(9)

6.10 The MidasRegMch Layer

MidasRegMch introduces all refinements specific to the random access variant of the MIDAS register file. All register indexes are trivially replaced with references to the specifier byte fields within the MIDAS instruction. As the value-range expressible by a byte field exceeds the register domain, events detecting out-of-domain accesses are used. This is in contrast to many microprocessor ISAs in which the instruction field may only express the register domain [Hit98].

A refined event constructing a successful NOP instruction from this layer is subsequently shown in Figure 7 of Section 8.

6.11 Model Summary

A list of the refinement stages within the general layers described previously is given in **Table 1**.

Table 1. Refinement summary.

Layer	Events	POs	Description
StateMch	2	4	Define top-level single event
StateMchR1	5	2	Split event by status value
StateMchR2	6	1	Split RUNNING events into behavior for valid and invalid insts
StateMchR3	7	5	Split good-inst into exec and halt instructions
StateMchR4	8	2	Split exec-inst into success and failure outcomes
ControlFlow	9	11	Refine inst to instArray/instPtr. Split exec-inst into OK/fail/bad-PC
ControlFlowR1	12	15	Split exec-inst into increment and control-flow
ControlFlowR2	13	3	Split flow-inst into conditional/vector available/not -available
ControlFlowR3	15	5	Split flow-inst into conditional true/false
ControlFlowR4	17	31	Split flow-inst-true into relative and absolute
RegArrayMch	20	17	Introduce data array. Split incr-inst into null-inst and those acting on data array
RegArrayMchR1	21	4	Refine operation-ok in reg-write to src-OK and destination-OK
RegArrayMchR2	35	70	Refine reg-write into fetch, move, single-op and dual-op instructions
RegArrayMchR3	35	18	Define reg indexes to be immediate data in inst
RegArrayMchR4	42	47	Refine fetch data into inst-ptr, immediate or memory
MemMch	45	24	Introduce memory. Split null-inst into null and insts acting on memory
MemMchR01	46	5	Fetch-mem insts refined to select element from mem via mem-index, guarded by readable flag
MemMchR02	48	6	Mem-write insts refined to select element from mem via mem-index, guarded by readable flag
MemMchR03	48	17	Refine mem-index to address stored in reg-array.
MemMchR1	48	20	Refine reg-indexes for mem-indexes as immediate data in instruction
MemMchR2	48	113	Refine all other reg-indexes as immediate data in instruction
MemMchR3	52	129	Refine Data-type elements to DataLong-type. Split fetch-immediate to byte/short/long
MemMchR4	56	57	Split mem-access insts into byte/short/long
MemMchR5	56	89	Refine mem to DataBytes. Define multi-byte data mappings for short/long accesses
ConFlagMch	59	32	Introduce con- flag. Split null to to null and con-flag-write. Refine flow conditionals to con-flag
ConFlagMchR1	59	16	Refine con-flag write to 2-input compare
ConFlagMchR2	59	4	Define compare src indexes as immediate data in instruction
ConFlagMchR3	63	5	Split compares into equals/greater-than/less-than
CalcMch	82	207	Split dual-op insts into add/subtract/multiply etc.
CalcMchR1	83	24	Split single-ops insts into casts
MidasMch	83	29	Define flow vector as reg data.

MidasMchR1	87	61	Eliminate fetch-immediate-short. Split move-inst into store and fetch.
MidasMchR2	84	0	Merge eliminated events into null-event
MidasMchR3	107	941	Merge instArray and mem into byte-array. Assign numerical values to insts. Fix inst and data sizes
MidasMchR4	108	8	Construct loader state machine
MidasMchR5	108	13	Split store-byte to io or memory. Eliminate fetch-bad-ptr condition
MidasMchR6	107	0	Merge fetch-bad-ptr into null-event
MidasStkMch	108	34	Refine direct addressing to datum/offset
MidasStkMchR1	107	433	Introduce stack-size. Refine reg indexes to stack-size relative.
MidasStkMchB2C	113	719	Refinements for B2C translation
MidasRegMch	109	517	Refine reg indexes to bytes within instructions
MidasRegMchB2C	109	1166	Refinements for B2C translation

Proof Obligations (PO) for all refinement stages are discharged using all of the Rodin proving tools. The increasing necessity for manual interaction during later refinement stages reflects the large number of hypotheses visible to the Rodin automatic proving tools at this point, thus requiring selection by the developer in order to achieve proof. A summary of the proof obligations and the method of discharge is given in **Table 2**.

Table 2. Proof Obligation summary.

Layer	**Automatic Discharge**	**Manual Discharge**	**Total**
StateMch	4	0	4
StateMchR1	2	0	2
StateMchR2	1	0	1
StateMchR3	5	0	5
StateMchR4	2	0	2
ControlFlow	25	6	31
ControlFlowR1	15	0	15
ControlFlowR2	3	0	3
ControlFlowR3	5	0	5
ControlFlowR4	11	0	11
RegArrayMch	17	4	17
RegArrayMchR1	4	0	4
RegArrayMchR2	69	1	70
RegArrayMchR3	18	0	18
RegArrayMchR4	45	2	47
MemMch	21	3	24
MemMchR01	5	0	5
MemMchR02	6	0	6
MemMchR03	17	0	17
MemMchR1	20	0	20
MemMchR2	111	2	113
MemMchR3	33	96	129
MemMchR4	29	28	57

MemMchR5	5	84	89
ConFlagMch	19	13	32
ConFlagMchR1	6	18	16
ConFlagMchR2	0	4	4
ConFlagMchR3	5	0	5
CalcMch	68	139	207
CalcMchR1	6	18	24
MidasMch	8	21	29
MidasMchR1	45	16	61
MidasMchR2	0	0	0
MidasMchR3	295	646	941
MidasMchR4	8	0	8
MidasMchR5	13	0	13
MidasMchR6	0	0	0
MidasStkMch	4	30	34
MidasStkMchR1	81	352	433
MidasStkMchB2C	411	308	719
MidasRegMch	142	375	517
MidasRegMchB2C	588	578	1166

7. Modeling of Data Elements

Modeling of an ISA within a formal environment highlights two common aspects of microprocessor design that greatly increase specification complexity: multiple interpretations of stored data and the fragmentation of large data elements across multiple smaller elements. For example the same thirty two bit long-word may be treated as an integer during an addition operation or a simple bit field during shift or bit-wise AND operations. This long-word will also be stored in memory as four contiguous 8-bit bytes, as illustrated by Figure 6.

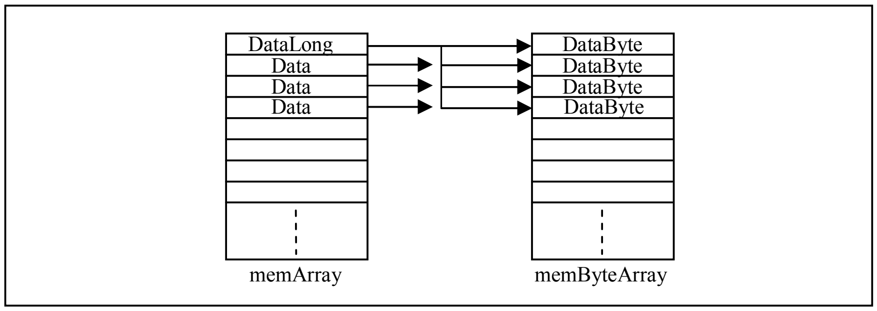

Figure 6: MemMch DATA/byte mapping

Figure 6 illustrates a significant side effect of this storage method for formal specification: modification of such a data element implies modification of the data elements mapped at subsequent locations. Such dualities are exposed to the programmer by the weak typing of the C language [KR88], and must therefore be captured by a model.

7.1 Modeling of Data Meaning

The possibility of multiple meanings of a stored data element make the basic Event-B types insufficient, and all data stored within the modeled machine is initially abstracted to a generic SET *DATA*. This allows simple data transfers within the machine to be modeled at an abstract level. For example, the event shown in Equation (10) specifies the transfer of data from memory to the register file during an abstract memory-fetch operation.

```
WHEN
  grd1: regIndex ∈ RegArrayDom
  grd2: memIndex ∈ MemArrayDom
THEN
  act1: regArray(regIndex) := memArray(memIndex)
```
(10)

However, if a stored data element is used to affect the behavior of the machine within the model, type translation functions must be introduced to explicitly state the relationship between *DATA* and its local meaning. In hardware design, and in C applications, this translation is implicit, potentially leading to ambiguity of behavior. A typical example is the translation between *DATA* and the integer set during storage and extraction of the PC, requiring the introduction of the mapping functions shown in Equation (11).

```
DataInt ⊆ ℤ
Data2Int ∈ DATA → DataInt
Int2Data ∈ DataInt → DATA
```
(11)

An example showing storage of the PC to the register file using this method is shown in Equation (12).

```
act1: regArray(regIndex) := Int2Data(instPtr)
```
(12)

7.2 Modeling of Data Size

Beyond the definition of the abstract *DATA* type, refinements must be introduced to capture the different data sizes used within a machine: 8-bits (byte), 16-bits (short) and thirty-two bits (long). Therefore *DATA* is recursively refined into three sub-sets, representing the three sizes:

```
CONSTANTS
  DataLong
  DataShort
  DataByte
AXIOMS
  DataLong ⊆ DATA        // Long can be contained by DATA
  DataShort ⊆ DataLong   // Short can be contained by Long
  DataByte ⊆ DataShort   // Byte can be contained by Short
```
(13)

Construction of these refinements is accompanied by refinement of the type-translation mappings discussed in 7.1:

```
DataByte2Int ∈ DateByte → 0..255
DataLong2Int ∈ DateLong → -2147483647..2147483647
```
(14)

7.3 Modeling of Data Fragmentation

The construction of a thirty two bit *DataLong* from four eight bit *DataByte* elements is initially specified by the following abstract mapping function:

$$\begin{array}{l} \mathrm{DataBytes2DataLong} \in \\ \quad \mathrm{DataByte} \times \mathrm{DataByte} \times \mathrm{DataByte} \times \mathrm{DataByte} \rightarrow \\ \qquad \mathrm{DataLong} \end{array} \tag{15}$$

Conversely, abstract record accessor mapping functions [EB06] are defined to specify the re-extraction of these *DataByte* elements:

$$\begin{array}{l} \mathrm{DataLong2DataByte0} \in \mathrm{DataLong} \rightarrow \mathrm{DataByte} \\ \mathrm{DataLong2DataByte1} \in \mathrm{DataLong} \rightarrow \mathrm{DataByte} \\ \mathrm{DataLong2DataByte2} \in \mathrm{DataLong} \rightarrow \mathrm{DataByte} \\ \mathrm{DataLong2DataByte3} \in \mathrm{DataLong} \rightarrow \mathrm{DataByte} \end{array} \tag{16}$$

The location of each *DataByte* within a *DataLong* must then be defined. In the AXIOM given in Equation (17), the *DataLong* implicitly constructed by *DataBytes2DataLong* is specified as four contiguous *DataBytes* within a generic array:

$$\begin{array}{l} \forall A,m,n,i,l \cdot A \in m..n+3 \rightarrow \mathrm{DataByte} \wedge i \in m..n \Rightarrow \\ \quad l = \mathrm{DataBytes2DataLong}(A(i) \mapsto A(i+1) \mapsto A(i+2) \mapsto A(i+3)) \end{array} \tag{17}$$

This AXIOM is complemented by that given in Equation (18), defining the *DataByte* accessed by each record accessor function:

$$\begin{array}{l} \forall a,b,c,d,l \cdot \\ a \in \mathrm{DataByte} \wedge b \in \mathrm{DataByte} \wedge c \in \mathrm{DataByte} \wedge d \in \mathrm{DataByte} \wedge \\ l \in \mathrm{DataLong} \wedge l = \mathrm{DataBytes2DataLong}(a \mapsto b \mapsto c \mapsto d) \Rightarrow \\ a = \mathrm{DataLong2DataByte0}(l) \wedge \\ \ b = \mathrm{DataLong2DataByte1}(l) \wedge \\ \quad c = \mathrm{DataLong2DataByte2}(l) \wedge \\ \quad\ d = \mathrm{DataLong2DataByte3}(l) \end{array} \tag{18}$$

Collectively these statements define the machine's byte-order, or endianess [HP03], as little-endian.

The arrangement of *DataBytes* within a *DataLong* allows similar statements to be derived as THEOREMS for the *DataByte* ordering within a *DataShort*:

$$\begin{array}{l} \forall a,b,s \cdot a \in \mathrm{DataByte} \wedge b \in \mathrm{DataByte} \wedge s \in \mathrm{DataShort} \wedge \\ s = \mathrm{DataBytes2DataShort}(a \mapsto b) \Rightarrow \\ a = \mathrm{DataLong2DataByte0}(s) \wedge \\ \quad b = \mathrm{DataLong2DataByte1}(s) \end{array} \tag{19}$$

7.4 Modeling of Memory-Mapped Data

The machine's memory system is initially modeled as a simple array mapping between an address range *MemDom* and the generic *DATA* set:

$$\mathrm{memArray} \in \mathrm{MemDom} \rightarrow \mathrm{DATA} \tag{20}$$

Refinement to a more precise specification of the memory system is performed in two stages. The *memArray* is initially refined to a new array of *DataLongs*:

$$\mathrm{memDataLongArray} \in \mathrm{MemDom} \rightarrow \mathrm{DataLong} \tag{21}$$

A gluing invariant establishes equivalence across the entire domain of the abstract array:

$$\forall x \cdot x \in MemDom \Rightarrow memArray(x) = memDataLongArray(x) \tag{22}$$

A second refinement step refines the mapping of each address to a *DataLong* by refining the memory itself to an array of *DataByte* in Equation (23), and a gluing INVARIANT in Equation (24) establishing the *DataLong*/*DataBytes* mapping described in 7.3.

$$memDataByteArray \in MemByteDom \rightarrow DataByte \tag{23}$$

$$\begin{aligned} &\forall x \cdot x \in MemArrayDom \Rightarrow memArrayDataLong(x) = \\ &DataBytes2DataLong(memDataByteArray(x) \mapsto \\ &\quad memDataByteArray(x+1) \mapsto \\ &\quad memArrayDataByte(x+2) \mapsto \\ &\quad memArrayDataByte(x+3)) \end{aligned} \tag{24}$$

Proof Obligations generated to prove the maintenance of this gluing invariant during memory-write operations are discharged by instantiation of (18) or (19) for all memory locations modified by that write.

8. Implementation Generation

Event-B and the Rodin tool are intended to support automatic generation of executable source code from sufficiently refined models [But06]. This functionality is not yet part of the current Rodin functionality, and therefore a plug-in extension [SDF03] was developed to support a sufficient subset of Event-B to support the VM project, translating to the C language [Wri09/2]. An example showing the final refinement of the MIDAS NOP instruction, and its translated C implementation is given in Figure 7.

```
NopOk
REFINES NopOk
ANY
    op
    opVal
    nextInstPtr
WHERE
    grd6: op :  DataSmall
    grd7: op =  mem(instPtr)
    grd5: opVal :  DataSmallNat
    grd2: opVal= DataSmall2Nat(op)
    grd1: opVal =  16
    grd3: instPtr <=  99994
    grd4: statusCode =  2
    grd8: nextInstPtr :  DataLargeNat
    grd9: nextInstPtr =  instPtr + 1
THEN
    act1: instPtr :=  nextInstPtr
END
```

```
/* Event5 [NopOk] */
BOOL NopOk(void)
{
    /* Local variable declarations */
    DataLargeNat nextInstPtr;
    DataSmall op;
    DataSmallNat opVal;

    /* Guard 1 */
    op = mem[instPtr];
    DataSmall2Nat(op,&opVal);
    if(opVal!=16) return BFALSE;

    /* Guard 2 */
    if(instPtr>99994) return BFALSE;

    /* Guard 3 */
    if(statusCode!=2) return BFALSE;

    /* Local assignments in actions */
    nextInstPtr = (instPtr+1);

    /* Actions */
    instPtr = nextInstPtr;

    /* Report hit */
    ReportEventbEvent("NopOk",5);
    return BTRUE;
}
```

Figure 7: Event-B event and derived C

Each event is translated to a separate C function returning a boolean signifying whether the event has been triggered. A function is generated to call all event functions in turn until an event is triggered, or signal if no event has been triggered at the end of the machine iteration (i.e. deadlock has occurred).

The calling function implicitly introduces determinism into models containing non-deterministic event triggering, as events are run in the same order, defined by their position in the Event-B model, Therefore in the case of multiple events being enabled precedence is always given to earlier events. The translator requires that precise values are assigned to all ranges and codes, and set membership is reduced to direct comparison operations. Range checking is performed to ensure that the implementing C type may contain numerical values and ranges. State variables are disallowed from the right side of actions, in order to prevent use after modification by preceding action-derived statements.

Guard statements are automatically evaluated for one of three possible interpretations: type definition of local parameters, assignments to local parameters, or conditional evaluations. Conditionals are implemented as negations of the basic comparisons enabling early returns from a function, and local parameter assignments are only calculated immediately prior to use. Thus execution is optimized and assignments are only evaluated in a valid context. Comments and instrumentation is inserted to provide traceability between the model and implementation.

9. MIDAS Demonstration

In order to test the constructed MIDAS VMs, a support environment is provided using conventional coding techniques. The environment provides a mechanism for download of binary images into MIDAS executable memory, and text output via a virtual console. These facilities are integrated with the automatically generated MIDAS implementations using conventional C development tools. Each MIDAS variant was built into 2 different calling environments: a standard environment used to run an error-free binary compiled from C, and a test environment used to execute a series of binary images coded in C, assembler or machine-code to exercise all the events-derived functions of the VM.

C compilers targeted at each MIDAS variant were provided via an appropriate GCC assembler and compiler back-end [Sta01]. Thus a hand-coded C test-suite could be compiled, loaded and executed, demonstrating the suitability of the MIDAS VM for supporting C programs. The test-suite, which was not developed using Formal Methods, is not a complete test of the C language, but includes the most common constructs and invokes the major executable fragments implemented by the compiler. An assembler-coded bootstrap performs segment initialization and machine shutdown. String initialization demonstrates the correct use of the read-only data region, length calculations test integer-based looping, and output to the virtual console tests character manipulation. Integer-to-string conversions test integer arithmetic and integer digit display via C switch statements test use of dispatch tables. Passing of function arguments and results test variable passing via stack pushes. Explicit tests exercise floating point arithmetic and casting. Integer field extraction functions test bit-wise shift and masking functions. Nested for and if statements test in-function nesting. Nested calls test deeply stacked function calls and returns.

10. Lessons Learned

Experience gained during the development of the generic VM model and MIDAS demonstrators allow future improvements to both Event-B and the Rodin tool to be suggested. Model development using multiple refinement steps requires considerable repetition of event guard and action sections. In larger models refinement can lead to small changes being difficult to identify amongst previously constructed functionality, and as the principle of guard strengthening allows additional guards to be legitimately introduced, erroneous guards may be introduced, potentially leading to unintended deadlock conditions. Such issues may be prevented and detected by improved tool support: more concise statement of model refinements (as now provided by the Event-B EXTENDS statement) and enabledness checking tools (as currently proposed).

Within the Rodin tool, the concept of a stored model database, indirectly editable via specialized tools, allows for efficient storage and traversing of model logic by tool extensions. Improvements in the ergonomics of the editing interface are required to achieve productivity similar to that of conventional text editing interfaces. Search, replace and pasting capabilities within a single view would allow faster, less error-prone development. Early versions of these tools are now available and should greatly enhance the development process.

As shown in Section 6.11, current Rodin automatic proving tools are vulnerable to scaling issues. A particular problem is the inability to select pertinent hypotheses from the large lists generated for large models, requiring developer intervention to select manageable sub-sets for PO discharge.

In common with other logical proving tools, the Rodin provers are susceptible to incorrect discharge of proof obligations due to accidental introduction of contradictory predicates. It is suggested that vacuity-checking techniques be introduced to defend against such conditions, as incorporated in other commercially available tools [Bee97].

11. Future Work

11.1 Model Checking

The VM model has been developed using incremental refinement techniques, with all generated POs discharged (as summarized in **Table 2**) in order to prove consistency between refinement steps. However, this analysis does not guarantee correctness of the model itself. In the case of the VM, checking against implicit deadlock states is of particular importance. Tools exist for the checking of model correctness [LB03], and use of these for checking of the VM model is desirable.

11.2 Architecture Management Tools

The construction of larger models would be enhanced by the development of high-level tools to assist in the understanding and navigation of deeply refined models containing large numbers of events, such as MIDAS. For example graphical display of a model's refinement hierarchy allowing the browsing and selection of particular events would greatly enhance productivity.

11.3 Prover Scaling Support

Relatively superficial enhancements to existing proving tools could be implemented to overcome the scaling issues already discussed and enhance automatic proving rates. For example, the ability for a developer to add programmable tactics, allowing pattern searching of available hypotheses and automatic addition of new hypotheses would be a powerful enhancement. Such techniques are particularly relevant to applications such as MIDAS, which contain many similar POs requiring repetitive application of relatively simple (currently manual) tactics.

11.4 Proof Inspection Tools

Current Rodin proving tools enable the automatic and semi-automatic discharge of the hundreds of POs generated by a model such as MIDAS: an essential capability if such Formal Methods are to be practical in this scale. However, only limited ability to review a discharged proof is currently available, although provided by other tools [BCo06]. This capability would allow the manual checking of proofs if required, and allow a developer to plan proof tactics during manual proving, by inspection of similar existing proofs. Tools listing the hypotheses and axioms used by a proving tool to discharge a PO could provide a useful first step towards a more complete proof description.

11.5 Automatic Test Generation

Formally derived models have been recognized as a possible basis for generation of testing criteria [UL07]. The VM application allows the opportunity for testing to be performed on a deployed machine, test inputs being provided by the loading of appropriate binary executables.

11.6 Full C Testing

The MIDAS ISA and compiler have been demonstrated to support a hand-coded example application: testing of the VM against a complete C test-suite is required to fully demonstrate the VM [She07]. The MIDAS ISA has been specified to demonstrate the Event-B modeling technique without regard to other metrics such as performance: expansion of the ISA to include additional performance-enhancing instructions and features, within the Event-B modeling paradigm discussed here, is possible.

11.7 Deterministic Model

The VM model currently allows for non-determinism under certain conditions. For example, two separate events are constructed to raise separate exceptions if either the source or destination are unavailable on an attempted data transfer, but the event triggered in the case of both being unavailable is not defined. Expansion of the model to deterministically enumerate all such conditions would yield a more precise specification of VM behavior under such error conditions, albeit at the cost of greatly increased model size.

12. Conclusions

Event-B allows the generic properties of binary Instruction Set Architectures to be captured in an abstract model, thus providing a re-usable template for the development of Formally Proved computing machines. The Event-B refinement process allows an incremental structure in this abstract model, maximizing its re-usability, and its concretization to a level sufficient for automatic conversion to a usable implementation. Constructed relationships derived within the model may also be used to guide specification of new ISAs.

The Rodin tool enables the management of the multiple refinement stages and Formal Proof analysis required for such a technique, and provides the capability for necessary implementation generation tools to be developed.

The technique has been demonstrated by the construction of such a model, and its refinement to multiple implementations in the form of Virtual Machines capable of running compiled binary images.

13. References

[Abr96]	Abrial,J-R: "The B-Book: Assigning Programs to Meanings", 1996
[ABH06]	Abrial,J-R Butler,M Hallerstede,S Voisin,L “An Open Extensible Tool Environment for Event-B”, Formal Methods and Software Engineering, SpringerLink, 2006

[AMD07] AMD Inc “128-Bit SSE5 Instruction Set”, 2007

[BCo06] B-Core “The B-Toolkit User Manual” B-Core (UK) Ltd,, 2006

[Bee97] Beer,I Ben-David,S “RuleBase: Model checking at IBM”, CAV, 1997

[BH91] Brock,B Hunt,W “Report on the Formal Specification and Partial Verification of the VIPER Microprocessor”, Proceedings of the Sixth Annual Conference on Computer Assurance, Systems Integrity, Software Safety and Process Security, 1991

[But06] Butler,M “Rodin Deliverable D16 Prototype Plug-in Tools”, http//rodin.cs.ncl.ac.uk, 2006

[Cas02] Caset,L “Formal Development of an Embedded Verifier for Java Card Byte Code”, International Conference on Dependable Systems and Networks, 2002

[Ecl09] Eclipse. Eclipse platform homepage. http://www.eclipse.org/, 2009

[EB06] Evans,N Butler,M “A Proposal for Records in Event-B” Formal Methods 2006, 2006

[EG07] Evans,N Grant,N "Towards the Formal Verification of a Java Processor in Event-B", Proceedings of the BAC-FACS Refinement Workshop, 2007

[Fox03] Fox,A "Formal Specification and Verification of ARM6", Theorem Proving in Higher Order Logics, SpringerLink, 2003

[GB90] Graham,B Birtwistle,G "Formalising the design of an SECD chip", Hardware Specification, Verification and Synthesis: Mathematical Aspects, SpringerLink, 1990

[HP03] Hennessy,J Patterson,D “Computer Architecture, A Quantitive Approach”, Morgan Kaufmann, 2003

[Hit98] Hitachi Ltd. “SH7707 Hardware Manual”, 1998

[Hun94] Hunt, W “FM8501: A Verified Microprocessor”, Lecture Notes in Artificial Intelligence Subseries of Lecture Notes in Computer Science, Springer-Verlag, 1994

[KR88] Kernighan,B Ritchie,D “The C Programming Language”, Prentice Hall, 1988

[KN01] Klein,G Nipkow,T "Verified bytecode verifiers", Foundations of Software Science and Computation Structures, SpringerLink, 2001

[LBSL97] Lapsley,P Bier,J Shoham,A Lee,E “DSP Processor Fundamentals” IEEE Press 1997

[Lee89] Lee,E “Programmable DSP Processors part I and II” IEEE ASSP Mag Oct 1988, Jan 1989

[LB03] Leuschel,M Butler,M “ProB: A Model Checker for B” FME 2003, SpringerLink, 2003

[LY99] Lindholm,T Yellin,F “The Java Virtual Machine Specification, Second Edition”, 1999

[MAV05] Metayer, C Abrial, J-R, Voisin, L “Rodin Deliverable 3.2 Event-B Language”, http//rodin.cs.ncl.ac.uk, 2005

[Pat07] Patterson,D “Computer Organization and Design: The Hardware/Software Interface”, Morgan Kaufmann, 2007

[Qia99] Qian,Z "A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines", Formal Syntax and Semantics of Java, SpringerLink, 1999

[Sch01] Schneider,S “The B-Method An Introduction”, Palgrave, 2001

[SDF03] Shavor,S D’Anjou,J Fairbrother,S “The Java Developer’s Guide to Eclipse” Addison-Wesley, 2003

[She07] Sherridan,F “Practical Testing of a C99 Compiler Using Output Comparison”, Software: Practical and Experience, Volume 37 Issue 14, 2007

[Spi89] Spivey,J.M “The Z Notation: A Reference Manual”, Prentice-Hall, 1989

[SM95] Srivas,M Miller,S “Formal Verification of an Avionics Microprocessor”, Langley Research Center, 1995

[Sta01] Stallman,R “Using and Porting the GNU Compiler Collection”, Free Software Foundation, 2001

[SSB01] Stark,R Schmid,J Borger,E. “Java and the Java Virtual Machine”, Springer, 2001

[UL07] Utting, M Legeard, B Practical Model-Based Testing – A Tools Approach, Morgan Kaufmann, 2007

[Wri08] Wright,S “Using EventB to Create a Virtual Machine Instruction Set Architecture”, Abstract State Machines, B and Z, SpringerLink, 2008

[Wri09/1] Wright,S. “MIDAS Machine Specification”, Bristol University http://www.cs.bris.ac.uk/Publications, 2009

[Wri09/2] Wright,S “Automatic Generation of C from Event-B”, Workshop on Integration of Model-based Formal Methods and Tools, 2009