A Demonstration of Specifying and Synthesising Hardware using B and Bluespec

Ian Oliver
Nokia Research Centre
Itämerenkatu 11-13
00180 Helsinki,Finland
ian.oliver@nokia.com

Abstract

Hardware development is complex and the cost of errors extremely high. The use of specification and verification technologies is critical to managing the correctness and complexity of the systems being developed. Bluespec is a declarative hardware description language (HDL) that solves a number of the problems associated with translating a specification to a synthesisable form for hardware implementation. Combining Bluespec development with formal specification with B offers a sophisticated verification environment with the associated reduction in development errors.

1 Introduction

It is well known that with the increasing cost of system development, complexity and the need to ensure that the systems produced are 'bug free' more sophisticated and declarative development methods are required. Declarative here meaning the ability to concentrate on the domain or problem space of the system without reference to the implementation, execution semantics and the associated architectural pollution that comes from making design and implementation decisions in the specification.

Specification languages such as B [1], Z [18], Alloy [17] etc and accompanying methods such as the B-Method (in the case of B) have been successfully used in industry for a number of years with successes primarily in the safety-critical software domain. Attempts have been made to use these languages for hardware descriptions but with varying degrees of success [12, 19, 15, 9, 4][1].

Hardware is different from software in a number of fundamental ways and thus poses problems that are not experienced in the software domain. For example, the notion of a variable in software often becomes a wire in hardware with very different semantics. Hardware, at least synchronous anyway, has the notion of a clock and overall hardware has implicit parallelism: a function in a piece of software takes input values and then computes one instruction at a time through a number of intermediate values the output value, whereas in hardware the input and output values are directly related in terms of time.

The use of formal specification in hardware development is well known and its necessity comes from the costs associated with hardware failures and bugs. The ability to construct a correct and validated specification long before the RTL description stage allows more freedom in architectural choices and more importantly the removal of certain classes of errors [10] from the design thus reducing the amount of non-exhaustive, complex and expensive simulation of RTL descriptions. One of the major problems with simulation and testing is that they are by their very nature non-exhaustive. Although systems with apparent large state spaces can be model checked [5], the tractable size of state space is insignificant to the whole [8]. Finding the interesting states to model check might also be impossible.

In this paper we describe the B specification language and the Bluespec hardware description language which together offer the possibility of solving the mix of declarative, abstract specifications with the efficient synthesis of hardware [14].

2 The Languages

We describe the two languages briefly by the use of a simple 'lift controller' specification in B and its possible translation or implementation in Bluespec. Figure 1 shows a pictorial representation of the system which consists of up and down buttons, a display showing the current floor and a mechanism for moving the lift between floors.

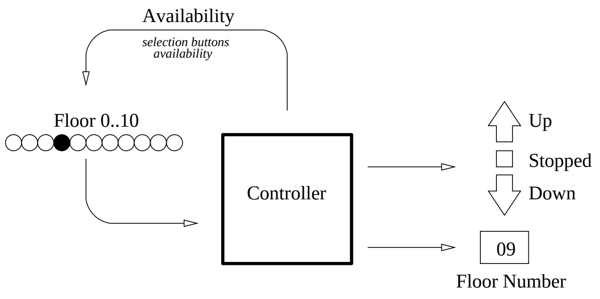

Figure 1: Simple Lift Controller

We specify a number of safety constraints on the system:

[1]Also refer to http://download.gna.org/brillant/docs/B-Bibliography/Bmethod-hardware_bib.html for a more complete bibliography regarding B and hardware development

- While moving the lift must not be ‘available’
- The lift can not move outside its designated floor range

2.1 B

B is a formal language based on algebraic specification and designed for modularity, refinement and eventual mapping to code. It is supported by a method called the B-Method and a variety of tool sets providing theorem proving[2], model checking and animation [11][3]

A specification consists of a number of related B machines which describe the system in a structured manner. Each machine consists of a number of variables describing the state space which values are modified through the application of operations and constrained through the use of an invariant.

```
MACHINE liftP
CONSTANTS topFloor
PROPERTIES topFloor = 10
SETS
 MOTORSTATUS =  UP, DOWN, STOPPED ;
 LIFTSTATUS =   AVAILABLE, NOTAVAILABLE
VARIABLES
 currentPosition, requestedPosition,
 availability, activity
```

The invariant constrains the state space through typing (lines 1..4 below), and by expressing safety properties (lines 5..14) such as the nonavailability of the control during movements (lines 13 and 14).

```
 1| INVARIANT
 2|  currentPosition : INTEGER &
 3|  requestedPosition : INTEGER &
 4|  availability : LIFTSTATUS &
 5|  activity : MOTORSTATUS &
 6|  ((activity = UP) =>
 7|    (currentPosition <= requestedPosition)) &
 8|  ((activity = DOWN) =>
 9|    (currentPosition >= requestedPosition)) &
10|  currentPosition >= 0 &
11|  currentPosition <= topFloor &
12|  requestedPosition >= 0 &
13|  currentPosition <= topFloor &
14|  ((activity = UP)=>(availability = NOTAVAILABLE)) &
15|  ((activity = DOWN)=>(availability = NOTAVAILABLE))
```

A B machine must always be initialised through an Initialisation clause which here states that the lift is available, stopped and on floor 0. This must be consistent with the invariant.

```
INITIALISATION
 currentPosition := 0 ||
 requestedPosition := 0 ||
 availability := AVAILABLE ||
 activity := STOPPED
```

Operations are specified in terms of a guard (actually a precondition) and action; if the guard resolves to true then the operation may take place. If the operation is performed then the actions are performed in parallel as an atomic unit. The proof obligations for an operation ensure that the resultant state of an operation conforms to the invariant; thus after any operation the system will always be in a consistent state with respect to the invariant.

[2]AtelierB (ClearSy), B-Toolkit (B-Core)

[3]ProB (M.Leuchel, University of Düsseldorf, Germany)

```
OPERATIONS
stoplift =
 PRE
  currentPosition = requestedPosition &
  not(activity = STOPPED)
 THEN
  activity := STOPPED || availability := AVAILABLE
 END ;
```

The above operation can only be fired when the current position of the lift is the same as that of the requested position *and* the activity of the lift is not stopped. If this operation fires under these conditions then the activity of the lift will be stopped and it made available. This is guaranteed through the verification process (ie: theorem proving) to be consistent with the invariant of the lift system.

The specification/development flow continues from the specification of a machine through its decompositions and their refinements; B has a strong notion of refinement which guarantees that all machines and their refinements are behaviourally consistent with each other.

2.2 Bluespec

Bluespec (Bluespec System Verilog)[4] is a rule based, declarative hardware specification language based on term rewriting. It is supported by a compiler that is capable of scheduling the set of rules without explicitly specified execution order. The compiler checks for shared variable update, calculates the execution order and guarantees synthesisable SystemVerilog. A large set of libraries for common hardware structures such as registers, FIFOs and other structures is provided.

A description written in Bluespec is divided into two parts: interface and module. Many modules can implement an interface and this is analogous to VHDL's entity-architecture concept. Each method described in the interface must be implemented in a module; by methods external entities may communicate with the module.

```
interface ILift;
 method Action request(Int#(8) f);
endinterface

module lift(ILift);
 method Action request(Int#(8) f)
  if (activity == Stopped);
   requestedPosition <= f;
    if (f != currentPosition)
     activity <= Moving;
 endmethod
endmodule
```

Bluespec supports through libraries a number of basic types such as Integer (of varying widths), Boolean, Queues (FIFO etc) and so on. Structures with unions and intersections can also be specified, though it is impossible to have recursive types (eg: linked list style

[4]www.bluespec.com

structures). Below we see the definition of two registers, one containing 8 bit integers and the other some user defined type:

```
1| Reg#(Int#(8)) currentPosition <- mkReg(0);
2| Reg#(MotorStatus) activity <- mkReg(Stopped);
```

Line 1 above is read: the location with name *currentPosition* supports the register interface with parameterised type 8-bit integer. *currentPosition* is bound to an instance of a module implementing that interface called *mkReg* with initial/reset value of 0.

As can be seen above we may have enumeration types sets of default operations (usually just equality) and bit representation also provided if necessary through the use of the deriving statement:

```
typedef enum  Up, Down, Stopped
        MotorStatus deriving(Bits,Eq);
```

The key structure in Bluespec is that of the rule. These are used to capture the internal workings of the hardware being described. The difference between a method and a rule is that a method is either used for querying or for setting the state of the system.

Rules take the form of a guard and action; when the guard is true then the rule can (or does) fire and the actions inside the rule are performed. Each rule is atomic and all items in the action body are performed in parallel; expressions are normally expanded out to some combinational logic structure.

```
rule stoplift
  ( currentPosition == requestedPosition &&
    activity != Stopped );

  activity <= Stopped;
  availability <= Available ;
 endrule
```

The above rule *stoplift* corresponds to the B stoplift operation shown earlier and is read: stoplift fires when the guard that the current position is the same as the requested position and activity is not Stopped, then activity is assigned the value Stopped and in parallel availability assigned the value Available.

Execution of an individual rule is atomic and is always completed in a single clock cycle - this is the basic term rewriting semantics of Bluespec. There are of course pragmatic concerns regarding how much logic can be fitted into a single clock cycle and this must be taken into consideration when synthesising - future versions of the compiler will perform analysis automatically for this and similar conditions.

The semantics of Bluespec is then further refined with the addition of a scheduler which states: all the rules that can fire, will fire in parallel on the same rising clock edge. This leads to the situation that if any wires or structures such as registers are updated then the updates must be mutually exclusive. If two rules are not mutually exclusive then the compiler will flag this as a warning and select one rule to fire over the other, that is a default priority mechanism exists.

The rules under the above conditions have a total order of priority such that if two rules conflict due to register writes or preconditions then one rule takes precedence over the other to conform to the term rewriting semantics. The priority of rules can be customised if needed.

3 Syntactic Mapping B to Bluespec

The structure of a Bluespec description is in the form of an interface declaration and module declaration which implements an interface. All Bluespec methods must be declared in the interface and implemented in the module. Many modules may implement the same interface. We take the approach that a method is derived from an operation that is used to communicate with the environment and a rule for any operation which modifies the internal state. We can demonstrate the concrete syntactic mapping given the following B operations:

```
moveDown =
PRE
 activity = DOWN &
 currentPosition > requestedPosition &
 currentPosition > 0
THEN
 currentPosition := currentPosition - 1
END ;

request(ff) =
PRE
 ff : INTEGER &
 ff >= 0 & ff <= topFloor &
 not(ff = currentPosition) &
 availability = AVAILABLE
THEN
 requestedPosition := ff ||
 availability := NOTAVAILABLE
END

av <-- is_available =
BEGIN
 av := availability
END ;
```

we infer the following by mapping the B precondition and action to their respective places as rules and methods in a Bluespec module declaration:

```
 rule moveDown(activity == Down &&
        currentPosition > requestedPosition &&
    currentPosition > 0 );
    currentPosition <= currentPosition - 1;
 endrule

 method Action request(Int#(8) ff)
  if (availability == Available);
  if ( (ff >= 0) && (ff<=topFloor) &&
     (ff!=currentPosition) &&
     (availability == Available) )
  begin
   requestedPosition <= ff;
   availability <= NotAvailable ;
  end
 endmethod

method LiftStatus is_available;
 return availability;
endmethod
```

There is a syntactic issue regarding the guards in that parameters to a method (or rule) can not be referred to in the guard, hence the slightly odd looking if statements - the first of which is the method's guard - above. Typing predicates (ff : $INTEGER$) in B are supported in the parameter definition: $Int\#(8)ff$.

For these examples we simply map B's integer to an 8 bit integer register and enumerated set similarly. More complex typing is possible but we do not cover these here in this demonstration.

The actual interface and module appear as below, we have used ellipses to denote where additional code will lie. Any type definitions relating to enumerations are declared before the interface:

```
typedef enum { Up, Down, Stopped }
   MotorStatus deriving(Bits,Eq);
typedef enum { Available, NotAvailable }
   LiftStatus deriving(Bits,Eq);

interface ILiftP;
 method Action request(Int#(8) f);
 method LiftStatus is_available;
 ...
endinterface

(* synthesize *)
module liftP(ILiftP);
 ...
endmodule
```

3.1 Type and Variable Defintions

We can now generalise this to the abstract syntax: enumerated types are always mapped at bit structures with the equality operated defined:

$$\begin{array}{l} SETS \\ \qquad S = E \end{array}$$

as: `typedef enum E S deriving(Bits,Eq)`

Variables are mapped from three B constructs:

$$\begin{array}{l} VARIABLE \\ \qquad V \\ INVARIANT \\ \qquad V : T \\ INITIALISATION \\ \qquad V := E \end{array}$$

as: `I(T) V <- M(E)`

where I is an interface definition that supports the given type T and M is a module that conforms to the interface. Typically we map variables to registers (although wires could be an optimisation) and types such as integer to some suitable range, eg: 8, 16 or 32 bits. Only the typing of the variable defined in the invariant is required to be mapped. It is possible to map further conditions from the invariant through the use of Bluespec assertion statements, we have not extensively addressed or investigated this at this time.

3.2 Operations

There are three basic kinds of mapping for B operations to Bluespec depending on whether these operations update the system, query it or attempt to do both. A further subclassification of this depends upon whether the operation is to be made available through the system's interface or not. The table below provides an overview of the mappings:

Parameter	Update	Return	Mapping
No	No	No	n/a
No	No	Yes	method
No	Yes	No	rule or method
No	Yes	Yes	method ActionValue
Yes	No	No	n/a
Yes	No	Yes	method
Yes	Yes	No	method Action
Yes	Yes	Yes	method ActionValue

Operations which change the state of the system, ie: those which input data into the system through parameters, are as follows:

$$\begin{array}{l} N(S) = \\ \qquad PRE \quad P \\ \qquad THEN \quad Q \end{array}$$

is mapped in the interface as:

```
method Action N(S);
```

and in the module as, where P also contains typing statements for the parameters in S.

```
method Action N(S) if (P);
      Q
endmethod
```

If there are additional conditions in P that further restrict the parameters then these must be placed in the body of the rule as Bluespec does not allow certain parameter constructs in the method guard. In these cases the parts of P (ie: P_s)which guard the operation based upon parameter values is mapped:

```
method Action N(S) if (P);
      if (Ps) then Q
endmethod
```

Operations which just query the state of the system and do not change anything are mapped similarly:

$$\begin{array}{l} R \leftarrow N(S) = \\ \qquad PRE \quad P \\ \qquad THEN \quad Q \end{array}$$

as before P includes typing statements for S, and Q assigns values to and gives a type the variables in R. In the interface this is mapped as:

```
method T N(S);
```

and in the module as:

```
method T N(S) if (P);
      return Q;
endmethod
```

Actions that return values as well as updating the state are mapped similarly to query operations but take the additional construction ActionValue which states this fact explicitly:

```
method ActionValue#(T) N(S) if (P);
      U;
      return Q;
endmethod
```

where U are the state updating statement and Q is the return assignment.

Operations which contribute to the update of the state but are not accessed from outside the system are mapped as rules; these are generally identified as operations which do not take parameters; these operations never produce return values:

$N =$
$\quad PRE \quad P$
$\quad THEN \quad Q$

is mapped as follows but in the module definition only - not in the interface:

```
rule N(P);
      Q;
endmethod
```

It is always possible that operations without parameters or return values which change the state of the system are not rule but methods. In our experience this occurs rarely but can be taken care of by explicitly stating in the mapping that this particular operation is mapped as a method. However we generally enforce that parameter- and return-less operations are always mapped as rules in any accompanying development method

3.3 Types

Concentrating more on the types available, Bluespec provides a rich environment for types which also includes a number of the SystemVerilog types. We briefly mentioned integers and their default mapping to integer registers.

B provides boolean and various forms of integer as basic types and richer structures via sets, records and tuples. These can be supported in Bluespec via `typedef` and `struct` constructs and tuples via `Tuple#(T1,T2)`. Structures such as union types would have to be custom architected. Sets and strings can be supported by the use of Bluespec's vectors and lists. Functions which define relationships can be supported by vectors of tuples of the domain and range types. Many of the decisions regarding the mapping of types is made during the architecting process from B to Bluespec, although as described above some classifications can be made as defaults.

Generally B's Boolean type maps to Bluespec's Boolean type, while integers are by default mapped to a suitable bit width in Bluespec. By default we tend to map to 32 bit signed Integers to conform to the AtelierB theorem prover's interpretation of Integer.

4 Semantic Mapping Issues

Both B and Bluespec are based upon action system semantics which are an extension of Dijkstra's guarded commands [6]. B uses atomic, interleaving actions; Bluespec is similar but with clock-scheduling and rule priority. Bluespec's term rewriting aspect is similar to that of B and is adequate for many functional correctness properties.

Clock scheduling of rule introduces timing properties and picks a deterministic route through the transitions described by the term rewriting. This guarantees that the system always reaches the states predicted by the term rewriting rules but may never visit some states allowed by this.

These can be summarised such that in B, one picks any enabled action and executes its body (this guarantees atomicity), while in Bluespec one executes any enabled rules atomically. This is further refined under scheduling: wait for a given clock event such as a rising clock edge and then execute the enabled set of rules. In practise a program in Bluespec will only 'visit' a subset of the states that a B specification can due to Bluespec's scheduler picking certain rules to fire rather than potentially any at random. Of course there are pragmatic conditions which again limit this such as the amount of hardware logic that can be reasonably fitted into a clock period, resource limitations such as the readability of registers and energy consumption.

Any given specification in B can be mapped to Bluespec but the scheduling policy and variable conflict rules will require additional guarding of the rules and methods in the Bluespec. There are three possibilities for refining the B specification to construct a 'correctly functioning´ Bluespec program:

- Change of rule priorities
- Splitting of rules
- Constraining rules

Of these, the first is a modification to the architecting process that generates the Bluespec, the others can be made inside the B specification itself.

Often just changing the rule priorities inside the Bluespec has an effect. Bluespec calculates the rule priorities from the syntactical ordering of the rules in Bluespec program. If these rules can be reordered then sometimes the conflicts are removed.

Splitting the rules in the specification is made to leverage more parallelism in the Bluespec. The sometimes has the effect of making the scheduling easier such that one rule may be split such that the preconditions are much simpler and are more focussed to the actual actions inside the rule - that is there is less potential conflict between the precondition and the mechanics of the actions themselves. When using this 'pattern' it often becomes clearer which parts of a rule are causing conflicts and thus isolating scheduling problems is much easier.

Further constraining the rules by adding additional precondition statements basically restricts the firing of the rules such that the scheduling is more deterministic and fixed. This may also be accomplished by the addition of additional variables which effectively encode the behaviour of the scheduler.

When these changes are made to the B specification we can under some circumstances use the refinement mechanism in B directly [2] to ensure that the modified B specification still adheres to the desired behaviour encoded in the most 'abstract' B machine. However, perculiarities in B and the semantics of refinement, especially in the latter cases of splitting of rules and encoding the scheduler through additional variables may cause problems with abstract forms of the additional rules and variables being required to be declared in the most abstract versions of the specification [16]. Typically one might have to introduce operations of the form:

```
op = BEGIN skip END
```

to the abstract specifications and variables declared but without use inside the specification. These have no real effect other than to clutter the specification somewhat.

5 Development Flow and Example Synthesis

The development flow of B is based around a top-down, strict refinement approach. Using this approach means that all the modelling is performed using B and a codegenerator makes the mapping to Bluespec automatically. However like any *model based* approach the process of architecting one model to another is complex and as we have shown already in some cases the mapping of types in one language to another depends very much upon the development context. It may be possible using more sophisticated analysis, model checking and refuters (such as Alloy) to analyse the range of some variables so that the smallest representation in terms of bits is made in the architecting process.

Development proceeds as per defined in the B-Method where an abstract specification is made and this is translated to Bluespec according to some defined architectural rules from the verified B specification. Two possibilities exist here, firstly that of following the traditional B refinement step where the specification is made more concrete and refinement proved between this more concrete specification and the more abstract, or, application of one of the patterns given earlier to solve a scheduling problem reported by the Bluespec compiler. This continues until we reach a suitable level of specification where the Bluespec is in a form that makes synthesis practical and sensible. This of course may occur on the first step of specification, but this depends upon the specification style used - certainly this is true if one adopts a more 'operational' style of writing specifications.

Given a Bluespec program generated from a B specification we can use existing synthesis tools to produce RTL and netlists for synthesis. Because we are working from a verified description this greatly reduces the development and testing times such that any increase in chip/area size and power consumption is annulled by the reduction in development, testing and debugging times. In the latter cases we are currently seeing approximately between -5% and +10% differences in power consumption and floor area compared to traditional VHDL based development for our current test designs. In the example the B and Bluespec are 100 lines each, the System Verilog RTL was 287 and this generated for a Xilinx Spartan FPGA: 48 FMAPS (24%), 5 HMAPS (5%), totalling 24 CLBs (24%) or for for an Altera Cyclone FPGA: 23 I/O ATOMs, 54 LUTs (1%), 62 ATOMs (2%) for logic resources, 234 inputs on ATOMs and 10769 nets. Figure 2 shows the top level circuit from the B-Bluespec development generated by Synplicity's SynplifyPro.

5.1 Notes on Automatic Translation

A suitable subset of B has not yet been established at this moment, although the subset B0 used in the *implementation machines* in B is guaranteed to be translatable to languages such as C, Ada and Java and this might provide a suitable starting place. Some constructs might be impossible to translate to hardware without great computational expense, for example recursive and dynamic structures such as linked lists. One issue is also that the B language is being superseded by EventB which has a simpler syntax and semantics and is overall more suitable for the specification of hardware based systems. While automating this

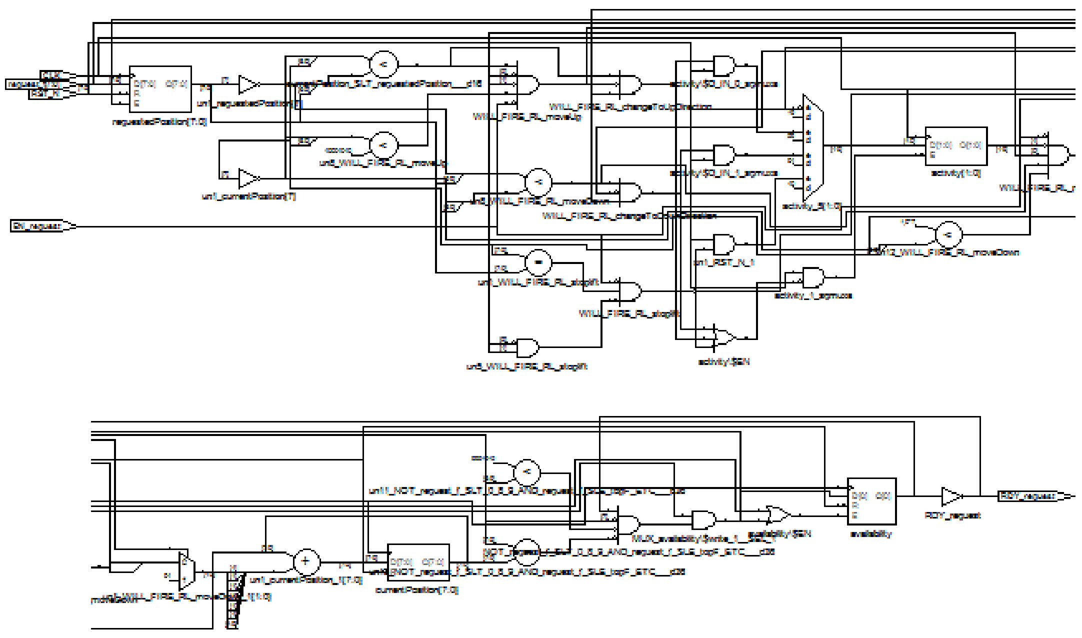
Figure 2: Top Level Circuit View

process is relatively simple, the move to EventB which at this point in time does not support all machine relationship constructs has delayed this. However as the EventB system is based on the Eclipse toolset with all the integration possibilities this provides then the absence of an automatic translation from B at this time is not considered to be a problem.

This of course is a concern to those who desire fully automatic translation rather than making the translation manually according to a set of rules: it is certainly possible that an automatic translation may itself introduce a number of very hard to find errors if too much trust is placed in the coding of the translation.

One possibility that is being investigated is that of mapping the Bluespec back to B - this abstraction process is much simpler than that of mapping B to Bluespec and provides two possibilities: firstly one can choose just to write in Bluespec and hide the B altogether. This has the advantage that it is possibly more palatable to those who consider formal specification languages to be 'difficult' in some manner.

The second possibility is to utilise the above Bluespec to B mapping with the development flow described in this paper. Through this one can analyse and gain confidence in the mapping (manual or automatic) between the B and the Bluespec. This is achieved by proving isomorphism between the B specification and the B abstracted from the Bluespec generated from the original B. This of course may not be completely trivial and requires further investigation.

6 Conclusions

The purpose of this paper was to *demonstrate* that hardware development via Bluespec is possible using the B specification language in a much more direct and natural way than a mapping to VHDL or SystemC would be [7]. B provides an environment where verification using theorem proving and validation through model checking/animation can be performed. The ability to produce correct software through B is well known [13], and here we can see that this is also possible with hardware.

This work is currently being employed and the approach validated by mapping existing B specifications of the interconnect layer of the Nokia **NoTA** hardware platform. This platform is a Corba/WebServices-like interconnect system for mobile devices and the IP-block being created is the mechanism to allow devices to talk directly with this network.

Traditional development flows first concentrate on simulation and then later synthesis. Bluespec guarantees that synthesis is always possible and the use of B ensures that the design is always verified at a much earlier stage than is possible now. This also means that the verification and validation of the system in inherent in the design itself - the correct by construction approach - this reduces by an order of magnitude logic errors. When the B is mapped to Bluespec we can be sure that the functional correctness of our system is preserved.

We are currently formalising the translation between B and Bluespec in the presence of more sophisticated architectural constructs: mapping of types to particular hardware structures and the integration with existing

IP blocks.

Additionally we can take into consideration existing work regarding the manipulation and analysis of action systems. One particular piece of work has been with the addition of fault tolerance patterns to B specifications [3].

7 Acknowlegements

To Antti Innamaa (Synplicity) for assistance with the synthesis tools, Rishiyur Nikhil (Bluespec, Inc.) and Joseph Stoy (MIT) for assistance with the Bluespec language and semantics. This research is being carried out as part of the EU funded research project: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

References

[1] J-R Abrial. *The B-Book - Assigning programs to Meanings*. Cambridge University Press, 1995. 0-521-49619-5.

[2] Ralph Back. *Refinement Calculus: a Systematic Introduction*. Springer-Verlag, 1998. 0387984178.

[3] Pontus Boström, Mats Neovius, Ian Oliver, and Marina Waldén. *Formal Transformation of Platform Independent Models into Platform Specific Models in MDA*. Technical Report 759, 2006.

[4] Jean-Louis Boulanger, Ammar Aljer, and Georges Mariano. *B/HDL, an experiment to formalizing hardware by software formal specifications.*. In *EDCC4, Fourth European Dependable Computing Conference, Parc des Expositions,Toulouse , France*. October 23-25 2002.

[5] J R Burch, E M Clarke, and K L McMillan. *Symbolic Model Checking: 10^{20} States and Beyond*. Information and Computation, 98(2):142–170, June 1992.

[6] E W Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[7] Stefan Hallerstede. *Parallel Hardware Design in B*. In Didier Bert, Jonathan P. Bowen, S. King, and M. Waldén, editors, *ZB'2003 – Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 101–102. Springer, Turku, Finland, June 2003. ISBN 3-540-40253-5.

[8] John Harrison. *Floating-Point Verification*. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 529–532. Springer-Verlag, 2005.

[9] Boulanger Jean-Louis. *BHDL - An example*. In *SCI 2004 - The 8th World Multi-Conference on Systemics, Cybernetics and Informatics, Orlando, Florida, USA*, volume IX, pages 150–155. International Institute of Informatics and Systemics, July 18-21 2004.

[10] M Larsson. *An Engineering Approach to Formal Digital System Design*. The Computer Journal, 38(2):101–110, 1995.

[11] Michael Leuschel and Michael Butler. *ProB: A Model Checker for B*. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003. ISBN 3-540-40828-2.

[12] Jean Mermet, editor. *UML-B Specification for Proven Embedded Systems Design*. 2004.

[13] Stephanie Motre and Corinne Teri. *Using B Method to Formalize the Java Card Runtime Security Policy for a Common Criteria Evaluation*.

[14] Michael Pellauer, Mieszko Lis, Don Baltus, and Rishiyur Nikhil. *Synthesis of Synchronous Assertions with Guarded Atomic Actions*. In *Formal Methods and Models for Codesign (MEMOCODE'2005), Verona, Italy, July 11-14*. 2005.

[15] Juha Plosila, Kaisa Sere, and Marina Waldén. *Component-Based Asynchronous Circuit Design in B*. Technical Report 377, Turku Center for Computer Science, December 1999.

[16] Michael Poppleton and Richard Banach. *Controlling control systems: an application of evolving retrenchment*. volume 2272, pages 42–61. Springer-Verlag Lecture Notes in Computer Science, 2002. ISBN 3540431667.

[17] The Alloy Project. *http://alloy.mit.edu*.

[18] J M Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, second edition, 1989. editied by C.A.R.Hoare.

[19] Tomi Westerlund and Juha Plosila. *Formal Modelling of Synchronous Hardware Components for System-on-Chip*. In *International Symposium on System-on-Chip*, pages 116–119. 2005.