A hardware/software codesign framework for developing complex embedded systems using formal model refinement.

Conference Paper · January 2004

Source: DBLP

CITATIONS
0

READS
166

4 authors, including:

Colin Snook
University of Southampton
138 PUBLICATIONS **1,521** CITATIONS

SEE PROFILE

Stefan Hallerstede
Aarhus University
79 PUBLICATIONS **1,743** CITATIONS

SEE PROFILE

Thierry Lecomte
ClearSy System Engineering
54 PUBLICATIONS **418** CITATIONS

SEE PROFILE

A hardware/software codesign framework for developing complex embedded systems using formal model refinement *

N. S. Voros
INTRACOM S.A.
254 Panepistimiou str., 26443
Patra, Greece

C. F. Snook
School of Electronics and
Computer Science
University of Southampton
Zepler Building, SO17 1BJ
United Kingdom

S. Hallerstede
KeesDA S.A.
Parc Equation, 38610 Gieres
France

T. Lecomte
ClearSy S.A.
Europarc de Pichaury 13856 Aix en Provence
France

Abstract

The approach proposed in this paper introduces a hardware/software codesign framework for developing complex embedded systems. The method relies on formal proof of system properties at every phase of the codesign cycle. The key concept is the combined use of UML and B language for system modeling and design, and the seamless transition from UML specifications to B language. The final system prototype emerges from correct-by-construction subsystems described in B language; the hardware components are translated in VHDL/SystemC, while for the software components C/C++ is used. The outcome is a formally proven correct system implementation. The efficiency of the proposed method is exhibited through the design of a case study from the telecom domain.

1 Introduction

Modern electronic systems tend to become more and more complex, supporting a range of functions in hostile environments like automobiles, railways and airplanes. The advent of such systems in more and more applications has led to a new category of systems called SoCs (Systems-on-a-Chip). The increasing competition and the market pressure have created the need for products of high reliability with short time-to-market. Thus, traditional development techniques, where the system development was relying on the experience of highly qualified engineers, are no longer adequate. The complexity of modern systems requires methodologies and supporting tools to deal with the increasing market requirements.

In this context, the next paragraphs introduce a codesign approach where system properties are formally proven throughout system design. The proposed codesign approach relies on

*This work has been performed in the framework of the IST project PUSSEE (IST-2000-36103). The PUSSEE project is partly funded by the European Commission. The authors would like to acknowledge the contributions of their colleagues from AB Volvo (Sweden), Nokia Corporation (Finland), Intracom S.A. (Greece), ClearSy S.A. (France), University of Southampton (United Kingdom), KeesDA S.A. (France) and University of Paderborn (Germany).

the combination of UML and B language, and is supported by appropriate tools enabling the seamless use of the two languages.

The rest of the paper is organized as follows: Section 2 presents the rationale of the proposed framework, while Section 3 explains its key concepts. In Section 4 the method toolset is described, and in Section 5 a real world case study is presented. Finally, Section 6 concludes by presenting an overview of the main paper concepts and future work.

2 Rationale & existing work

The main idea described in the next sections is to produce fully functional system models that are formally proven to be correct, and based on them to produce automatically the hardware and the software parts of the system. The approach presented relies on the combined use of UML and B language.

The UML [1] is a visual object oriented modeling notation for object oriented systems. Translation to a formal notation that has adequate tool support, such as B [2] enables a model to be formally verified. Formal models are also amenable to animation which allows early validation of requirements. These verification and validation processes are not available in the UML even if annotated constraints are added in the UML constraint language, OCL [3]. However, translation from unrestricted UML models is problematic because B language is not object oriented and contains write access restrictions between components in order to ensure compositionality of proofs. There are several approaches reported for mapping between UML to B language, e.g. [4]. The key difference in our approach is that we specialize UML in order to ensure that the resulting B is amenable to verification. To overcome write access restrictions we provide a translation from many classes (a package) to a single B component. We also provide UML mechanisms to support an event style of modeling and decomposition by supporting translation to a variant of the B language called Event B [5].

The use of B language for designing complex systems has already been reported in literature [6, 7]. As opposed to them, the approach presented in this paper proposes the use of B language as part of a unified hardware/software codesign framework that supports formal proof of system properties throughout the various codesign phases.

For the design of hardware system parts, the B language has been applied to circuit design. In [8], Event B is used to specify and refine a circuit but the approach does not provide a translation into a hardware description language. We have extended their work in Event B, while we have developed and used a BHDL (B-to-Hardware Description Language)[1] translator to generate SystemC and VHDL. The approaches in [9, 10] model circuits close to an implementation level. The first one requires that the model uses basic logic gates which are modeled in terms of B machines; the second one allows higher data types like integers in their models, while it introduces new structuring mechanisms into B which mirror those of VHDL closely. Our approach differs from all three in that it does not use an explicit representation for the system clock.

3 Proven electronic system design using formal proof of system properties

The approach presented in this paper is outlined in Figure 1. The main design phases of the proposed framework are described in the sequel.

System Modeling using a graphical formal notation. For that purpose a specialized profile of

[1] BHDL is a trademark of KeesDA S.A.

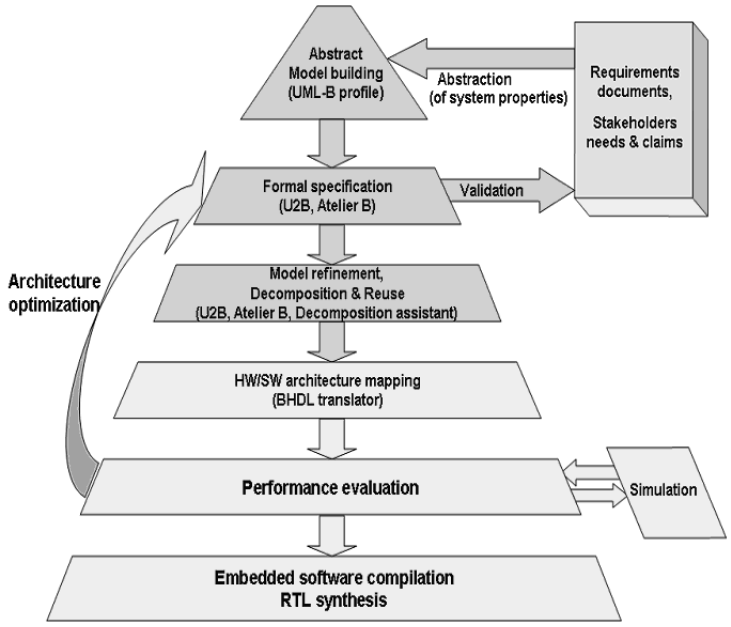

Figure 1: An overview of the proposed codesign framework

UML, called the UML-B profile, has been defined in order to allow designers to employ UML for defining system models and their properties during early design stages (upper part of Figure 1). The outcome of this process is a set of system models that contain formal descriptions of system properties in a B compliant manner.

Formally proven to be correct refinement where the initial system is gradually refined and each refinement is proven to be correct. The latter is guaranteed through the discharge of all the proof obligations generated (each proof obligation represents a system property that must not violated during refinement).

System decomposition into subsystems. Each subsystem can be mapped either to hardware or to software. At a specific refinement level, where the system representation is accurate enough, the system is decomposed into subsystems. The emerging subsystems and the communications between them are described in Event B and are produced automatically. Each subsystem can be further refined until a fully functional subsystem is reached.

Hardware/software allocation (lower part of Figure 1) takes place through direct translations of subsystems using appropriate translators. The software parts are implemented in C/C++, while the hardware parts of the system are described in VHDL/SystemC. It is important to mention that in both cases, the code produced stems from system models that are formally proven to be correct.

During the last stages of system design, the subsystems produced are simulated together in order to verify their integrated behaviour, while the overall system performance is evaluated. Based on the performance evaluation results, alternative architectures can be explored.

4 The supporting toolset

The codesign framework proposed is supported by a set of tools that allow system designers to produce systems that are composed of correct-by-construction subsystems. The tools cover all the codesign stages from specification down to implementation.

4.1 U2B translator

The U2B translator [11] converts UML-B models into B components. Translation from UML-B into B is necessary to gain access to other tools in the toolset.

In many respects B components resemble an encapsulation and modularization mechanism suitable for representing a class. However, to ensure compositionality of proofs, B imposes restrictions on the way variables can be modified by other components (even via local operations) leading to corresponding restrictions on the relationships between classes. Only hierarchical class association structures could be modeled using this translation. A second option translates a complete UML package (i.e. many classes and their relationships) into a single B component. The instances, attributes, associations and operations of the classes are represented in the same way but are collated into a single B component. This option allows unconstrained class relationship structures to be modeled but no operation calling is possible because all operations are within the same B component (a further restriction for proof reasons). However, if we view the operation bodies as declarative specifications of behaviour and defer design issues, such as how that behaviour is allocated to operations, this is not a severe restriction on the UML modeling.

Since B language is not object oriented, class instances must be modeled explicitly. Inheritance may be used to define the instances of a class as a subtype of another class. Attributes and associations are translated into variables whose type is a function from the class instances to the attribute type or associated class. For example a variable instance class A with attribute x of type X would result in the following B component:

MACHINE	A_CLASS
SETS	A_SET
VARIABLES	A, x
INVARIANT	A : POW(A_SET) & x: A $\rightarrow$ (X)
INITIALISATION	A := {} || x := {}
...	

The multiplicity of an association determines the type of function (partial, total, injection, surjection or bijection) used in its modeling. Attribute types may be any valid B expression that defines a set.

UML-B clauses are used to attach constraints to UML entities. For example, a UML-B clause, INSTANCES, may be used to define the instances of a class as an enumerated set. UML-B clauses can also be attached to other modeling entities such as states in a statechart. If a package component mapping is used, UML-B clauses that apply to the complete package may be used.

The behavior modeled on a class diagram is given by the specification of operations and invariants. Since we wish to end up with a B specification we use a notation based on B for these constraint and action definitions. To B, we add the object oriented conventions of implicit self referencing and the use of the dot notation for explicit instance references. This is illustrated in the examples below.

Figure 2: Translation of operations

Invariants may be specified in a UML-B clause attached to the relevant UML entity. Where applicable, U2B will automatically add universal quantification over all instances of a class or a hypothesis to complete the invariant.

Operations need to know which instance of the class they are to work on ('self'). The translation adds a parameter *thisCLASS* of type *CLASSinstances* to each operation. This is used as the instance parameter in each reference to an attribute or association of the class. However, in event systems, operations represent events that occur rather than called operations. Hence, events cannot have parameters. Therefore if a class operation has parameters (including 'self') they are translated into a non-deterministic selection substitution such as:

ANY thisCLASS, p1 WHERE
thisCLASS: <class instances> & p1: <type>
THEN ...'.

For event systems, operation guards and actions are specified either in a textual format attached to the operation, on the transitions of a statechart attached to the class or by a combination of both composed as a simultaneous specification.

In the example in Figure 2, *set_y* might have the following precondition and semantics, which would be translated as shown in the right hand column:

Precondition	SELECT
i > y.bx	i > bx(y(thisA))
Semantics	THEN
y.b_op(i) ||	b_op(y(thisA)) ||
IF y.bx<100 THEN	IF bx(y(thisA))<100 THEN
out:=FALSE	out:=FALSE
ELSE	ELSE
out:=TRUE	out:=TRUE
END	END
	END

Statechart Behavioral Specification

A statechart representation of behavior may be used. The U2B translator combines the behavior described by a statechart with any textual operation semantics. The collection of states in the statechart is used to define an enumerated set that is used in the type invariant of a state variable. The state variable is equivalent to an attribute of the class and may be referenced elsewhere in the class and by other classes. Statechart transitions define which operation call causes the state variable to change from the source state to the target state. To associate a transition with an operation, the transition's name must match the operation name. Additional guard conditions can be attached to a transition to further constrain when it can take place. All transitions cause the implicit action of changing the state variable from the source state to the target state. Additional actions can also be attached to transitions. The translator finds all transitions associated with an operation and compiles a SELECT substitution of the following form:

SELECT statevar=sourcestate1 & transition1_guards
THEN statevar:=targetstate1 || transition1_actions
WHEN statevar=sourcestate2 & transition2_guards

THEN statevar:=targetstate2 || transition2_actions
etc
END
|||
<textual operation specification>

Refinement

UML-B defines a "REFINES" relationship between packages. The B component produced from a package, R_{n+1}, that refines a package, R_n, will be a refinement instead of a machine and will have a "REFINES R_n" clause. If variables in R_{n+1} have the same names as those in R_n, B will assume that they are the corresponding refined versions of the abstract variables. Note however, that the UML representation of the variable could be very different in the refining package from that in the refined package. For example, the state variable defined by a statechart could be the refinement of a class attribute. Where the variables have different names (and possibly not a one-to-one refinement relationship) a gluing invariant must be specified in the refining class. This must be given in an INVARIANT clause within the refining package (or one of its classes).

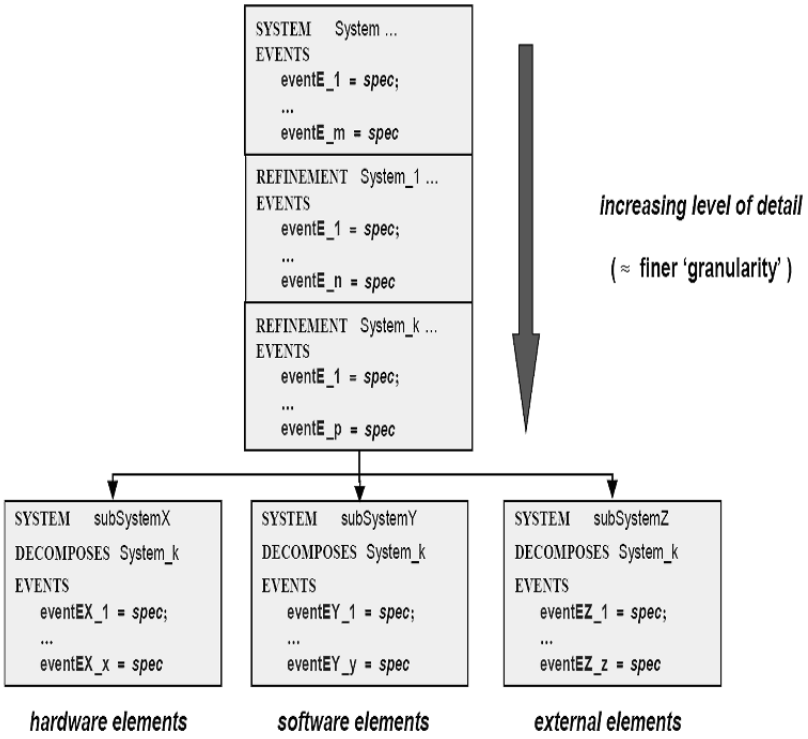

Figure 3: System decomposition and hardware/software allocation

4.2 Atelier B

The use of B language for describing and proving system properties between successive model refinement is crucial for the specific approach. The total number of proofs required, even for small systems, is usually big and difficult to handle without the support of an appropriate tool.

For the automation of proving process Atelier B[2] tool is employed. It is mainly composed of static analyzers that including (a) a type checker, which is responsible for the syntactic and

[2]Atelier B is a trademark of ClearSy S.A.

the semantic verification of a B component, (b) a B0 checker, which performs verification of system models described in B0 language[3] [2], and (c) the project verification analyzer which performs global verifications among system components in order to control the overall system architecture.

Additionally, Atelier B includes proof tools that allow formal proof of the successive B model refinements, where the proof obligations required can be proven either automatically or interactively. More specifically, the proof tools available from Atelier B include: (a) The automatic generator of proof obligations from the components in B, (b) the rule base manager (the rule base includes more than 2200 rules), (c) the automatic prover which discharges automatically most of the proof obligations, (d) the interactive prover which is used when the automatic prover has failed and (e) the predicate prover which demonstrates rules added by the user. Finally, Atelier B supports translation of B implementations to C/C++ for the software parts of the system under development.

4.3 Decomposition assistant

The decomposition assistant[4] automates the decomposition process [12] outlined in Figure 3. Decomposition is precisely the process by which a certain model can be split into various component models in a systematic fashion. As a result, the complexity of the proving process is reduced while the emerging subsystems can be implemented using different technologies. The communication among the components is automatically generated by the decomposition assistant. Communication consistency is based on the exchange of events among the subsystems and can be formally refined until the final communication scheme is reached.

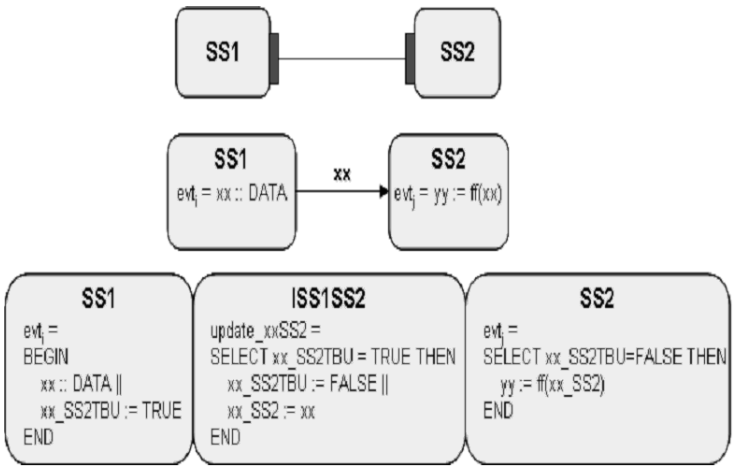

Figure 4: Communication between subsystems, as it is generated by the decomposition assistant

As described in Figure 4, during the decomposition process each variable is allocated to one and only one subsystem and references in other subsystems are copies that need to be updated. The lower part of the figure describes the modules generated by the decomposition assistant: SS1 and SS2. ISS1S2, which contains the communication primitives between the two subsystems, can be further formally refined in order to reach a fully functional protocol implementation.

System decomposition [14] is based on a decomposition profile defined by the designer. Decomposition assistant uses the decomposition profile as input and automatically produces the description of each subsystem along with the variables required for describing the selected com-

[3]B0 is a subset of B language, which is used only in implementations to ensure that they can be directly translated in C/C++.

[4]The decomposition assistant has been developed by ClearSy S.A. in the context of IST PUSSEE Project [13].

$tr(x := E)$	=	$x \Leftarrow E$;
$tr(S\|\|T)$	=	block begin $tr(S)tr(T)$ end block;
$tr(S;T)$	=	block signal $y : type_y$; begin
		$[y' := y'']tr(S)$
		$[y := y'']tr(T)$
		end block;

Table 1: VHDL Translation rules

munication scheme. Reuse of formally proven protocol descriptions through a protocol library is also applicable.

4.4 BHDL translator

The BHDL subset of the B language is similar to traditional (procedural) B. Structurally the main differences are the presence of input and output for a module, and the restriction of the operation clause to one substitution. Variables of BHDL machines are split into the categories INPUTS, OUTPUTS, and VARIABLES. The first two, the *ports* of the design, are externally visible with the obvious meaning. The other variables are local to the design. The OPERATION clause of a BHDL machine contains a single substitution describing the behavior of the design. Type and constant declarations may not be made in this kind of BHDL machines. They must be made in dedicated BHDL machines. This facilitates porting to different target languages.

BHDL data types are restricted to BOOL, INTn, and UINTn. In addition, enumeration types can be used and arrays. These types are contained in a basic BHDL machine, called *BHDL.mch* that must be imported in BHDL machines. In SystemC the types correspond to bool, sc_int<n>, sc_uint<n>, and enumeration types and arrays. This choice of data types facilitates portability to other hardware description languages. Arrays are represented as total functions in BHDL and may not be synthesizable after translation. If the design resulting from translation has to be synthesizable, it may be necessary to modify the design first by refinement. For simulation, however, this is not necessary. In practice, we found that most produced VHDL descriptions were readily sythesizable.

The substitution language of BHDL is a subset of the substitution language of B. BHDL machines are cycle accurate models of hardware which can be represented by a design on register transfer level in hardware description languages like VHDL or SystemC. In fact, register transfer level is not strictly enforced because some BHDL constructs and expressions correspond to the behavioural level. This is convenient for simulation. For performance analysis, a higher abstraction level (and earlier translation in the design process) would be useful. This work is under way for the transaction level of SystemC. The substitution language of BHDL comprises assignment "$x := E$", simultaneous substitution "$S||T$", sequential substitution "$S;T$", and conditional substitution "IF B THEN S ELSE T END". Arrays can only be assigned as a whole, i.e. assignments of the form "$x(k) := E$" are not possible. The reason is that tracking intermediary signals created in the translation would be complex and error prone, whereas the price of the incurred restriction in practice is very low. In this article we use lambda expressions to represent array values. So an array assignment has the form: "$x := k.(k \in K|E) \cup k.(k \in L|F)$", where $dom(x) = K \cup L$. The right-hand side may contain more than two lambda expressions.

Registers are inferred from variables declared in the VARIABLES clause of a BHDL machine depending on their use. Two sets *read* and *write* are calculated for a BHDL machine. Inputs declared in the machine must be contained in *read*, outputs in *write*. Variables that are contained in *read* and *write* are translated into registers. Input and output variables are translated into corresponding ports, and all remaining variables into wires.

Formally the translation into hardware description languages $tr(S)$ of S is based on the

MACHINE
DELAY
SEES
BHDL
INPUTS
din
OUTPUTS
dout
VARIABLES
quadneg, *buffer*
INVARIANT $neg \in UINT4 \wedge buffer \in 1..8 \rightarrow UINT4$
INITIALISATION
$buffer := \lambda x.(x \in 1..8|0)$
OPERATION
BEGIN
$neg := 15 - buffer(1)$;
$dout := neg$
END||
$buffer := \lambda x.(x \in 1..7|buffer(x+1)) \cup \lambda x.(x \in 8|din)$
END

Figure 5: BHDL machine

before-after predicate $prd_x(S)$ of a substitution S. For assignment in VHDL, simultaneous and sequential substitutions are described in Table 1 [15].

process (clock, reset) begin
if reset $='1'$ then
for k in 1 to 8 loop
$\text{buffer}_0(\text{k}) \leftarrow 0$;
end loop;
elsif clock'EVENT and clock $='1'$ then
$\text{buffer}_0 \leftarrow \text{buffer}_1$;
end if;
end process;

Figure 6: VHDL description of a register

Identifiers *clock* and *reset* must not be used in BHDL machines. Correspondingly named signals for use with registers are produced by the translation. Figure 5 shows a B implementation of a buffer that delays its input by 8 clock cycles. Variable buffer is initialized to an array of zeros. The translation generates a set of registers with reset for variable buffer (see Figure 6).

The state transition specified by the substitution is translated into a combinatorial circuit. We have left away the block statements to save space. Inputs and outputs are referred to by their original names. Signal names ending in n, with $n > 1$, refer to intermediate signals; suffixes *0* and *1* are used to model registers as shown above. The design shown in Figures 6 and 7 is sythesizable. Thus, so is the original BHDL machine in Figure 5 from which the VHDL code was produced by automatic translation. The BHDL machine could be further refined, e.g. by implementing the subtraction on bit level. In general, we prefer to allow as many high level constructs as possible, to allow earlier translation and analysis in the development process. For this reason we are also working on a translator to SystemC transaction level.

$neg_2 \leftarrow 15 - buffer(1)$;
$dout \leftarrow neg_2$;
process $(buffer_0, din_0)$ begin
for x in 1 to 7 loop
$buffer_1(x) \leftarrow buffer_0(x + 1)$;
end loop;

Figure 7: Combinatorial part of design

5 Case study: Specification and design of a telecommunication application

HIPERLAN/2 protocol [16] provides data rates of up to 54 Mbits/sec for short range (up to 150m) communications in indoor and outdoor environments. Typical application environments are offices, homes, exhibition halls, airports, train stations and so on. Figure 8 outlines the protocol architecture.

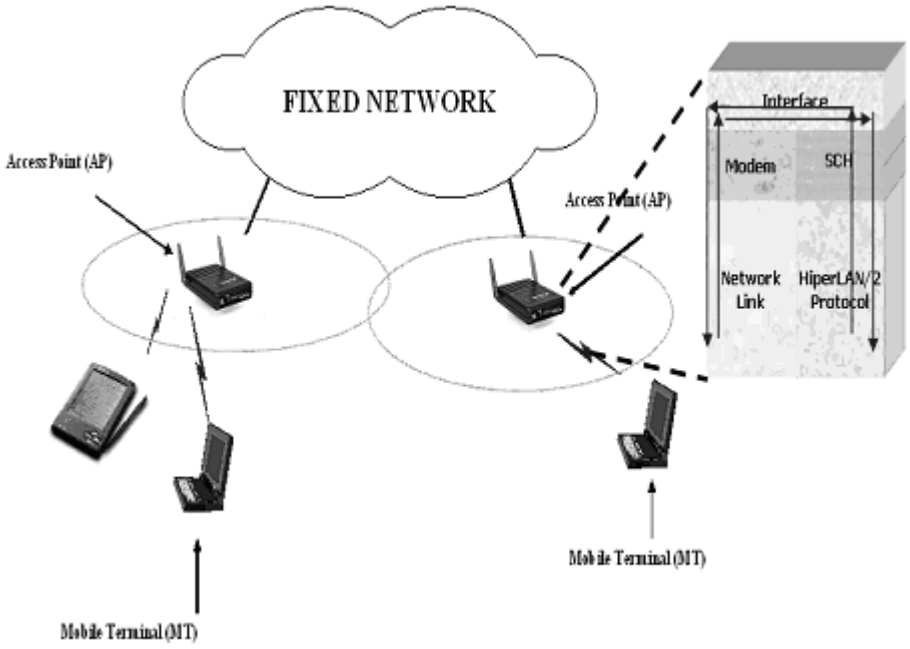

Figure 8: An overview of HIPERLAN/2 architecture

The system under design is part of the access point system and consists of the AP scheduler and the modem. The next paragraphs describe the design of the specific case study following the primitives introduced in the previous sections.

5.1 System specification using UML-B profile

In Figure 9 part of the overall system specification using UML-B profile is presented and corresponds to the SCH box depicted in Figure 8. The main parts of the Access Point Scheduler [16] include: *AP_SCHEDULER* which is responsible for the design of a MAC frame; *TRAFFIC_TABLE* that describes the next frame's logical channel entries required, according to the resource requests; *FRAME_INFO* that decides the number of information elements (IEs) and the number of blocks required (each block contains three IEs, the number of idle IEs and the

number of padding IEs); *DECISION* module that contains the decision algorithm used and *FCH* that contains the resource grants for the FCCH channel.

AP_SCHEDULER_1 is a formal refinement of the initial *AP_SCHEDULER*. The "imports" arrows refer to the corresponding keyword of B language and represent B modules containing part of *AP_SCHEDULER* functionality.

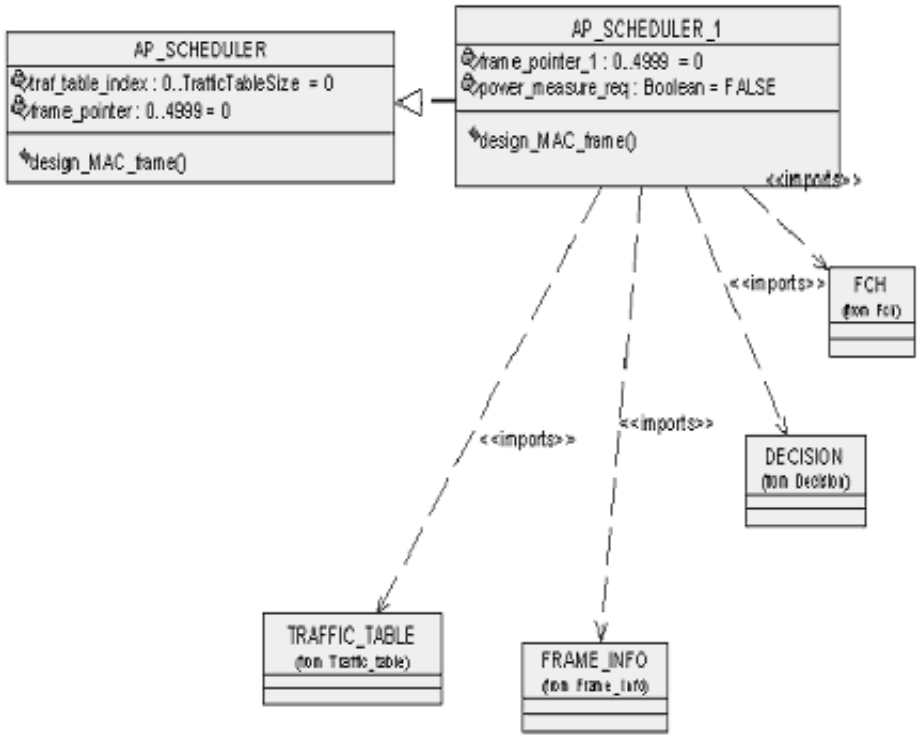

Figure 9: Description of HIPERLAN/2 Access Point scheduler using UML-B profile

5.2 Formally proven model refinement

In order to formally prove the correctness of the specific refinement, we used the U2B translator and Atelier B. Figure 10 delineates the B code produced by U2B translator for the *AP_SCHEDULER* class. *AP_SCHEDULER_1* class was also translated to B and the required proof obligations for the specific model refinement were generated using Atelier B.

The the refinement process revealed 3.353 proof obligations; 3.106 of them (92,6%) were automatically proven using Atelier B's automatic prover, while 247 proof obligations (7,4%) were proven using the interactive prover of Atelier B.

A similar process was followed for the rest of the system, including the modem part described in Figure 8. A fully implementable system model, described in UML-B profile, was finally produced where all the generated proof obligations were fully discharged.

5.3 System decomposition

Having a fully functional system description that was formally proven to be functionally correct, the next step was to use the decomposition assistant described in Section 4.3 to produce the hardware and the software subsystems. Part of the decomposition profile used is presented in Figure 11. This profile contains the number and the names of the subsystems specified by the designer, as well as variable allocation. Communication is deduced from that description,

using the default communication protocol for accessing data. These communication protocols are likely to be extended.

```
MACHINE AP_SCHEDULER_CLASS
/*" U2B3.6.4 generated this component from Class AP_SCHEDULER "*/
CONSTANTS
        AP_SCHEDULER
PROPERTIES
        AP_SCHEDULER = 1..n
VARIABLES
        traf_table_index,
        frame_pointer
INVARIANT
        traf_table_index : AP_SCHEDULER --> 0..TrafficTableSize  &
        frame_pointer : AP_SCHEDULER --> 0..4999
INITIALISATION
        traf_table_index :: AP_SCHEDULER --> {0} ||
        frame_pointer :: AP_SCHEDULER --> {0}

OPERATIONS
 design_MAC_frame (thisAP_SCHEDULER) =
BEGIN
    traf_table_index, frame_pointer:(traf_table_index:0..TrafficTableSize &
         traf_table_index<=2+6*(NumberOfMTsPlusBroadcast-1) &
         frame_pointer:0..4999)
  END

END
```

Figure 10: The initial B code produced for the AP_SCHEDULER

The system under design was decomposed in two subsystems: *SS_SCH* subsystem which corresponds to the functionality of the system UML-B model of Figure 9, and the *SS_MODEM* subsystem which contains the part of the initial system that contains the HIPERLAN/2 modem functionality.

5.4 Hardware/software allocation & implementation

The final step of the design process was the hardware/software allocation and the generation of final code. The final realization for the *SS_SCH* subsystem was implemented in software. More specifically, 2.291 lines of C code were generated using Atelier B's C code generator and the final software has been tested on ARM7 TDMI.

The *SS_MODEM* subsystem was implemented in hardware. For the hardware description BHDL translator was used in order to produce VHDL code from the available formally proven description.

6 Conclusions & Future work

In the previous sections we presented a detailed description of a hardware/software codesign approach for developing correct-by-construction embedded systems. The method introduced relies on formal refinement of system models, using a combination of UML and B language. The final system consists of formally correct implementations of C/C++ for the software parts and, VHDL or SystemC descriptions for the hardware components.

One important topic not covered in this paper is the formal proof of real time properties during system design, which is fully addressed in [17]. Additional improvements include further development of the UML-B profile and corresponding updates to the U2B tool[5]. A version is being produced that accepts XMI [19] as input and is independent of any particular UML

[5]The current version runs as a script within Rational Rose [18].

```
THEORY DECOMPOSITION IS
        SS_SCH=(Receive, Transmit, flush, assemble);
        SS_MODEM=(IQ_EN, RESET, ENDOP, NOP, CFG, RX)
END
&
THEORY SYNCHRONISATION END
&
THEORY ALLOCATION IS
        Allocate(newi, SS_SCH);
        Allocate(received_pointer, SS_SCH);
        Allocate(source_address, SS_SCH);
        Allocate(destination_address, SS_SCH);
        Allocate(src_ptr, SS_SCH);
        Allocate(current_sar_length, SS_SCH);
        Allocate(dest_length, SS_SCH);
        Allocate(buffer, SS_SCH);
        Allocate(valid_items, SS_SCH);
        Allocate(cell_arrived, SS_SCH);
        Allocate(cell_sent, SS_SCH);
        Allocate(PHY_Mode, SS_MODEM);
        Allocate(Valid, SS_MODEM);
        Allocate(Slot_Num, SS_MODEM);
        Allocate(PDU_Type, SS_MODEM);
        Allocate(EndM, SS_MODEM);
        Allocate(RPT, SS_MODEM);
        Allocate(Datatype, SS_MODEM);
        Allocate(EndT, SS_MODEM);
        Allocate(Enable, SS_MODEM);
        Allocate(EndR, SS_MODEM);
        Allocate(FrameNumber, SS_MODEM)

END
```

Figure 11: The decomposition profile for the telecom case study

tool, and will run within the Eclipse [20] open development platform. This will enable better integration of U2B with the B proof tools, which are also being ported to Eclipse. Finally, it is planned to extend the BHDL translator to higher level descriptions of circuits. It is expected that this combined treatment will also allow us the automation of many steps in lower level refinements.

References

[1] J. Rumbaugh, I. Jacobson & G. Booch, *The Unified modeling Language Reference Manual*, Addison-Wesley, ISBN 0-201-30998-X, 1998.

[2] J-R. Abrial, *The B Book: Assigning programs to meanings*, Cambridge University Press, 1996.

[3] J. Warmer, A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

[4] P. Facon, R. Lelau and H.P. Nguyen, *Combining UML with the B formal method for the Specification of database applications*, Research Report, CEDRIC Laboratory, Paris, 1999.

[5] ClearSy. Event B Reference Manual. Version 1.0. 2001. Available at: http://www.atelierb.societe.com/ressources/evt2b/eventb_reference_manual.pdf

[6] J. Draper et al, *Evaluating the B method on an avionics example*, Research Report, Proceedings of Data Systems in Aerospace (DASIA) Conference, 1996.

[7] C. Snook, L. Tsiopoulos, M. Walden, *A Case Study in Requirement Analysis of Control Systems using UML and B*, Proceedings of International Workshop on Refinement of Critical Systems, Methods, Tools and Developments, 2003.

[8] J-R. Abrial, *Event Driven Electronic Circuit Construction*, Available at: http://www.atelierb.societe.com/ressources/articles/cir.pdf

[9] J-L. Boulanger et al, *Formalization of Digital Circuits Using the B Method*, Proceedings of 8th International Conference on Computer Aided Design, Manufacture and Operation in the Railway and Other Advanced Mass Transit Systems, 2002.

[10] W. Ifill et al, *The Use of B to Specify, Design and Verify Hardware. In High Integrity Software*, Kluwer Academic Publishers, 43-62, 2001.

[11] C. Snook, M. Butler, *Final tool extensions for integration of UML and B*, Technical Report D4.1.3, Project IST-2000-30103 PUSSEE, 2004.

[12] J-R. Abrial, *Event Model Decomposition*, Available at http://www.atelierb.societe.com/resources/articles/dcmp3.pdf

[13] PUSSEE Project. Available at: http://www.keesda.com/pussee, 2003.

[14] T. Lecomte, J. R. Abrial, F. Badeau, C. Czernecki, D. Sabatier, C. Snook, *Abstract modeling: System level modeling and refinement in B*, Technical Report, Project IST-2000-30103 PUSSEE, 2003.

[15] KessDA. BHDL User Guide. Preliminary Version. Available At: http://www.keesda.com/pussee/bibliography.htm

[16] ETSI, *Broadband Radio Access Networks BRAN; HIPERLAN Type 2; Data Link Control (DLC) Layer Part1: Basic Data Transport Functions*, Technical Report ETSI TS 101 761-1 v1.1.1. 2000.

[17] A. Krupp, W. Mueller, *Refinement and verification of real time properies*, Technical Report D4.3.2, Project IST-2000-30103 PUSSEE, 2003.

[18] IBM Rational software. Available at: http://www.rational.com/, 2004.

[19] XMI. Available at : http://www.omg.org/technology/documents/formal/xmi.htm, 2004.

[20] Eclipse. Available at : http://www.eclipse.org/, 2004.