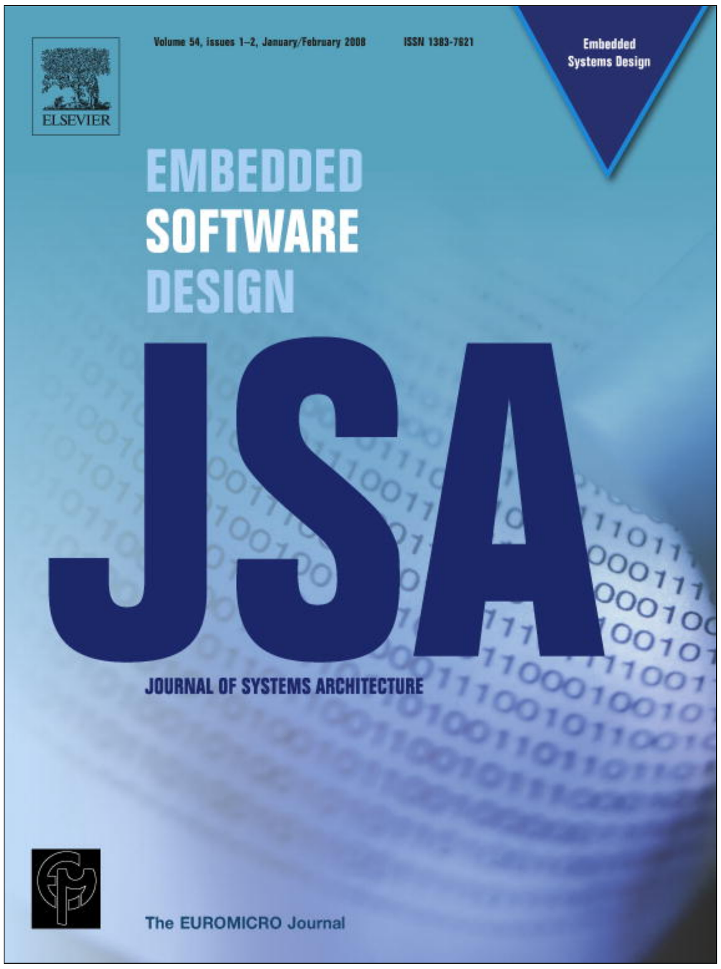
Volume 54, issues 1–2, January/February 2008
ISSN 1383-7621
ELSEVIER
Embedded Systems Design
EMBEDDED SOFTWARE DESIGN
JSA
JOURNAL OF SYSTEMS ARCHITECTURE
The EUROMICRO Journal

Available online at www.sciencedirect.com
ScienceDirect
Journal of Systems Architecture 54 (2008) 287–304
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

System level design of telecom systems using formal model refinement: Applying the B method/language in practice

Konstantinos Antonis [a], Nikolaos S. Voros [b,*]

[a] Department of Informatics and Computer Technology, TEI of Lamia, 3rd km Old National Road Lamia – Athens, 35100, Lamia, Greece
[b] Department of Communication Systems and Networks, TEl of Mesolonghi, Ethniki Odos Antiriou Nafpaktou, Varia, Nafpaktos 30300, Greece

Received 31 August 2005; received in revised form 29 June 2007; accepted 2 July 2007
Available online 21 July 2007

Abstract

The increasing complexity of modern telecommunication systems is one of the main issues encountered in most telecom products. Despite the plethora of methods and tools for efficient system design, verification and validation phases are still consuming significant part of the overall design time. The proposed approach outlines the use of the B method/language for producing correct-by-construction implementations of telecommunication systems. The method described is supported by appropriate tools that automate the process of proving that system properties are maintained during the various design stages. The feasibility of the latter is evaluated in practice through the design of a real world telecom application, borrowed from the domain of wireless telecommunication networks.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Formal methods; Formal verification; Correct by construction systems; System level design

1. Introduction

During the last years, the advances in modern telecommunications have resulted in the appearance of complex telecom systems that offer high quality services to the end users. The design and development of such systems is based on embedded hardware and software components, combined together, in order to achieve the overall system functionality. Each component is a system itself, usually complex, so the design of such systems is not a trivial task. As a result, telecommunication companies and system houses require effective system design methodologies and tools supporting their product line in order to stay at the leading edge.

The current situation regarding telecom system design in general is, that the methods are insufficient, informally practiced, and weakly supported by formal techniques and tools. Regarding system reuse, the methods and tools for exchanging system design data and know-how within companies are ad hoc and insufficient. In that context, system designers come across new challenges including:

- The development of high quality products that target to a highly competitive market.

* Corresponding author.
E-mail addresses: k_antonis@teilam.gr (K. Antonis), voros@teimes.gr (N.S. Voros).

1383-7621/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2007.07.001

- Decrease of time-to-market despite increased functionality, diversity and complexity.
- Demand for decreasing cost to face continuous market price erosion.

The approach presented in this paper delineates the concept of formal model refinement in system design. The main concept conveyed is that in order to deal with the aforementioned design challenges we need to (re)consider the design practices currently used, under a different perspective. Design experience has taught that the main bottleneck in system design processes is the stage of verification/validation. The proposed approach relies on the B method/language, which is a method for specifying, designing and coding complex systems. The method deals with the main aspects encountered during system design cycle, focusing on formal proof of system properties. In this way, the time spent for validation/verification is significantly reduced.

The rest of the paper is organized as follows: Section 2 provides an introduction to formal languages, while Section 3 describes the rationale of the work presented in the paper. Sections 4 and 5 outline the use of the B method/language for system level design, and the tool that supports it respectively. In Section 6, the design details of a telecom protocol for wireless systems is presented; Section 7 presents an evaluation the B method/language, based on the selected case study, while Section 8 concludes by presenting an overview of the main paper concepts and the future work.

2. Existing formal methods and languages

A *specification* can be regarded as a description that is intended to be as *precise*, *unambiguous*, *concise and complete* as possible in the context of its specific application [8]. A *formal specification* is a specification written in a formal language where a *formal language* is either based on a rigorous mathematical model or simply on a standardised programming or specification language [10]. Due to its individual application, a formal specification can be (partly) *executable*. In most cases, formal specifications are for a mental execution by code review and for passing the specification around to members in a design team. Generally, only subsets of formal specification languages, e.g. of Z and VDM, are machine-executable. A *formal method* implies the application of at least one formal specification language. Formal methods are often employed during system design when the degree of confidence in the prescribed system behaviour, extrapolated from a finite number of tests, is low. Moreover, they are frequently applied in the design of ultra-reliable as well as complex concurrent or reactive systems. Formal specifications can be classified with respect to their specification style. Here, we can identify mainly two different classifications. One is mainly due to the field of programming languages; the other one comes from general systems specification. In the rest of the section, we provide an overview of the most representative languages for formal specification.

Z was developed by the Programming Research Group at Oxford University and accepted as a BSI standard in 1989 [3]. It is a specification language based on set theory with no official method. Object-Oriented and real-time extensions to Z are available as Object-Z and Timed Communicating Object-Z, respectively. There are conventions and practices to use Z as a model-based language. Though Z permits various specification styles, the state-based approach has been found the most convenient one in many applications. Z comes with a deductive system in order to reason about specifications. It is based on typed set theory and first order predicate logic. Invariants can be associated with a global state. Invariants relate pre- and post- conditions for operations. The wider acceptance of Z in recent years has advanced the available set of tools. Tools for assistance with proof, and tools for translating Z specifications to programming languages, are available. However, a first high-level Z specification is usually not machine-executable.

The Vienna development method (VDM) is a formal specification method with the model-based specification language VDM Specification Language (VDM-SL) [11]. VDM was initially developed for the formal description of PL/I at the IBM laboratory in Vienna. The VDM method considers the verification of stepwise refinement in the systems development process, i.e. data refinement and operation decomposition. VDM-SL is conceptually similar to Z. VDM specifications are based on logic assertions of abstract states (mathematic abstraction and interface specification). In contrast to Z, VDM uses keywords in order to distinguish the roles of different components while these structures are not explicit in Z. As with Z specifications, VDM specifications are usually not machine-executable. VDM supports the specification process by a mental execution with paper and pencil. However, proof assistance tools and tools for executing subsets are available.

Computational tree logic (CTL) was defined as a branching-time temporal logic for model checking. Several variations of CTLs are known for practical applications: CTL, ACTL and CTL*. All CTLs are future-oriented. Only some approaches extend CTL with past modalities. CTL formulae express information about states or state transitions [13].

Temporal logic of actions (TLA) is temporal logic-based theory providing a logic for specifying and reasoning about concurrent and reactive systems [15]. TLA+ is the language for writing TLA specifications. The corresponding tool for mechanically checking TLA proofs is TLP (TL Prover), which is based on the Larch theorem Prover (LP) and a BDD-based model checker. TLA supports the specification of refinements and checks properties like fairness. A TLA specification is a list of formulae. A TLA formula is constructed by negation, conjunction, disjunction, implication and equivalence including the existential quantifier and predicates.

Calculus of communicating systems (CCS) [17] specifies a system as a set of asynchronously running processes performing, possibly non-deterministic, actions. CCS allows processes to be guarded by actions (*action-prefixing*). Processes are called agents in CCS and actions are referred to as the ports of an agent.

Communicating sequential processes (CSP) is conceptually similar to CCS [9]. CSP specifies a system as a set of asynchronously running processes acting on events. Processes communicate values (resp. events) via channels. To check CSP specification for properties, a satisfiability relation *sat* was introduced that allows CSP specifications to be checked with respect to a set of traces, i.e. specification vs. implementation.

CIRcuit CALculus (Circal) is a process algebra for the formal verification of digital hardware including asynchronous hardware [16]. Circal defines a set of core operators and a set of derived laws. The laws are based on the semantics of the core operators using a labelled transition system and equivalence relations. Circal compares to CCS. In contrast to CCS (and CSP), Circal supports simultaneous actions and multiway composition.

Finally, the B method, which stands for a language, a method and associated tools, is based on the hierarchical stepwise refinement and decomposition of a problem. After initial informal specification of requirements, an abstraction is made to capture, in a first formal specification, the most essential properties of a system. This top-level abstract specification is made more concrete and more detailed in steps, which may be one of two types. The specification can be refined either by changing the data structures used to represent the state information and/or by changing the bodies of the operations that act upon these data structures. Alternatively, the specification can be decomposed into subsections by writing an implementation step that binds the previous refinement to one or more abstract machines representing the interfaces of the subsections. In a typical B project, many levels of refinement and decomposition are used to fully specify the requirements. Once a stage is reached when all the requirements have been expressed formally, further refinement and decomposition steps add implementation decisions until a level of detail is reached at level B0, where code can be automatically generated for Ada and C/C++. B processing tools, like Atelier B from Clearsy [1], are advanced theorem provers with code generation, which automatically provide theorems, i.e. proof obligations.

3. Rationale

In order to deal with the increasing verification complexity, there are attempts to apply formal methods for verifying system properties during every design phase. The lack of mature tools and a complete method for developing complex systems are the main reasons that prohibit the use of formal methods in industrial environments. On the other hand, the ever increasing complexity of modern systems has led to increased design times, despite the fact that the time-to-market window is stringent in domains with increased competition, like telecommunications. One possible solution for decreasing time-to-market would be the reduction of time spent during the validation and verification stages. In that direction, we have adopted the use of the B method for designing a real world application borrowed from the telecommunication domain. Among the formal languages presented, the B method/language appears to be the most appropriate for the design of a commercial product. The rationale is that it is accompanied by a method that sufficiently covers all the design phases of a product, while there are mature tools that support the B based design throughout all the design stages.

Among the languages described in the previous section, the B language appears to be mature enough for the design of complex systems, while there are commercial tools that can support effectively the design process. The B method/language has already been applied for the design of complex systems [5,19], where the main focus is on system analysis aspects. Regarding the design of telecommunication products, the B method/language has been rarely used. In that context, the rest of this paper presents the experience gained from the design of a telecommunication system which constitutes a real world application, in an attempt to use effectively formal methods as part of an existing design flow. The design of the system relies on the B method/language. The initial system specifications were available in textual form, and based on them the design team involved in the specific case study has developed from scratch B abstract machines that have been used for the development of a fully functional system. The Appendix B at the end of the paper illustrates a representative example to help reader understand how the B method/language can be used for deriving invariants and preconditions for telecom system specifications. Additionally, the next sections provide detailed descriptions of the actual design problems encountered during the development of telecom systems with the B method/language.

4. System level design using the B method/language

4.1. System level design overview

The purpose of this section is to describe a method relying on the B language, for the design and development of complex systems. The key aspects of the approach presented are:

- the use of the B language for system specification and design,
- description of system properties in a formal way,
- formal proof between successive refinements,
- correct-by-construction implementation of the final system.

As described in Fig. 1, the first step of designing a system using the B method/language is to specify the system using B abstract machines (an introduction of the main concepts of the B method/language is presented in Appendix A). Having the B abstract machines at his/her disposal, the designer can start verifying the system. In the case where a verification error occurs, the designer must resolve it (either at the current or at the above specification level) and repeat the verification process making use of the modified system models.

When the verification process is completed successfully, the next step is system validation. If validation problems appear, the designer has to rework the system models in order to eliminate the problem. In the case of a successful validation, the designer can either refine the B abstract machines so as to produce more detailed system models, or decompose the system in the case the system model is too big to be implemented.

4.2. Formal model refinement

System models can be refined until they contain sufficient implementation details. As already explained, system specification starts with the definition of an abstract *machine* per subsystem, that describes the main properties of each subsystem in a formal way. The properties related to each system part are described using *invariants* and *preconditions* (for more information about invariants and preconditions please refer to Appendix A). In order to describe the properties of a system with sufficient preconditions and invariants, it is necessary for the designer (a) to be aware of the properties of the system under design, and (b) to have in depth knowledge of the B method/language so as to express them efficiently. An example of definition of an invariant defined in an abstract machine is the following statement:

```
ie_mac_id:(0...TrafficTableSize-1)+->0...255
```

which states that *ie_mac_id* variable is partial function from the *0...TrafficTableSize-1* set to the 0...255 subset of natural numbers. Every model emerging from the refinement of the specific abstract machine must be proven that does not violate the aforementioned invariant.

Formal model refinement guarantees that, during successive refinements of the abstract machines, every invariant and precondition defined will be fulfilled. The operations, the invariants and the preconditions produce a set of *proof obligations* that must be formally proven every time a system model is enriched with additional design details. The generated proof obligations between two successive

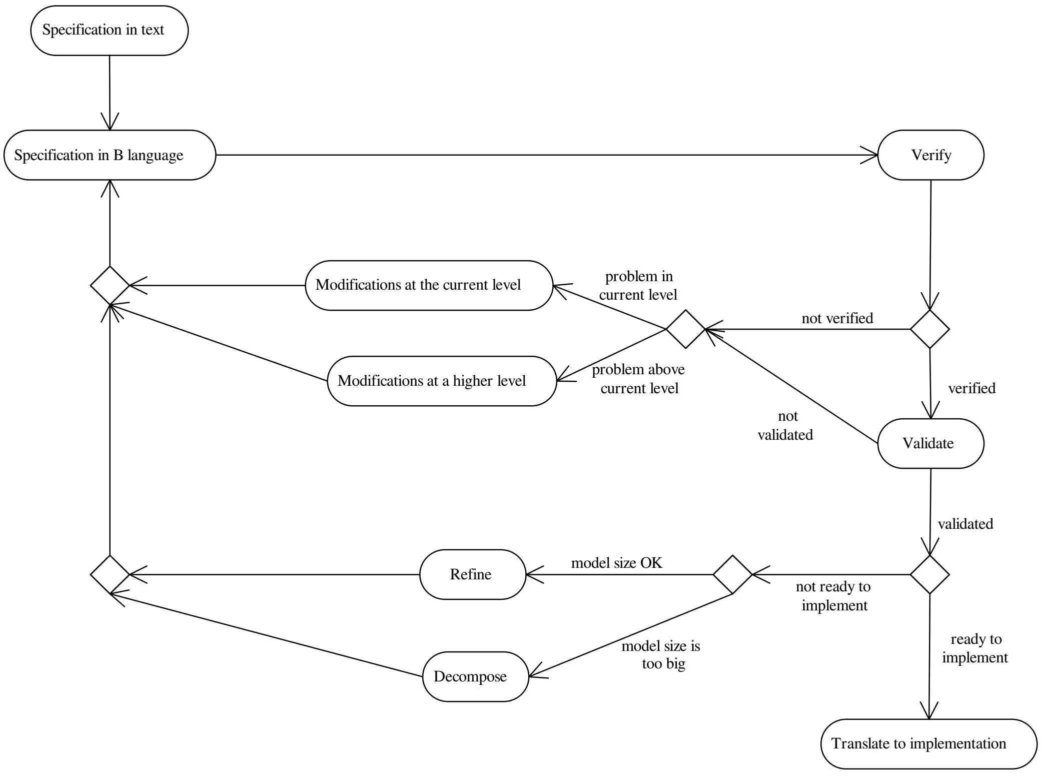

Fig. 1. System level design using the B method.

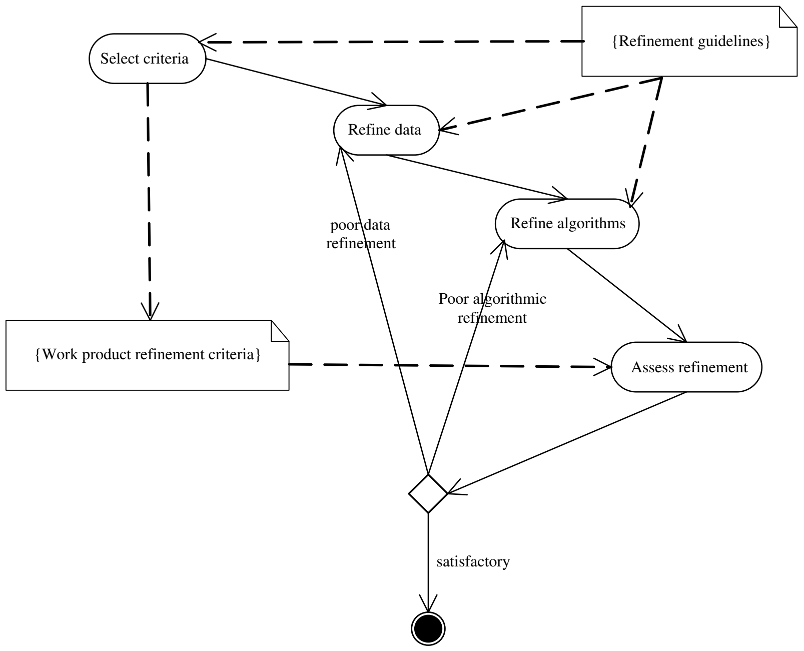

Fig. 2. Model refinement steps.

refinements have to be discharged before entering the next refinement level.

Fig. 2 outlines the process of formal model refinement. The designer is able to refine both data and algorithms during the refinement process. Usually, a set of refinement guidelines is available through out the refinement process, while the assessment of a specific refinement is strongly related to criteria derived from the nature of system under design.

The last refinement of a machine is called *implementation* and can use the specification of one or more abstract machines that can also be refined (with the use of the IMPORTS clause). As soon as all the proof obligations for all subsystems have been discharged and the specification models the requirements correctly, the designer has at his/her disposal an error free model. The next step is automatic translation from B0 [2] (a subset of B appropriate for code generation) to C/C++/Ada.

4.3. System decomposition

Depending on the complexity of the model refined, the number of proof obligations to be proven can significantly increase. In this case, it is usually preferable to decompose the specific refinement into smaller subsystems (see Fig. 1), by introducing new subsystems as abstract machines, that will be "imported" (with the use of the IMPORTS clause) in this level of refinement. Each subsystem is a system itself, with an initial set of requirements related to its functionality. The new subsystems can simplify the complex operations (the ones that create excessive numbers of proof obligations). As soon as the system has been decomposed into subsystems each subsystem is gradually refined leading to a subsystem implementation; the final system emerges from the recomposition of subsystems' implementations. In this *divide and conquer* design strategy, the initial system model is replaced by a set of subsystems' models and the proof obligations of the system are divided among its subsystems. The final system implementation emerges from the recomposition of error free (all initial proof obligations are fulfilled) subsystems' implementations. As it is presented in Section 6.3 system decomposition has been mainly used for model refinement in our case study, because of the large number of proof obligations.

5. Tool support

The use of B method/language for proving system properties during the design phase implies the use of a tool that is able to automatically handle the proof obligations generated during the refinement process. The approach presented in this paper relies on Atelier B[1] [1]. Atelier B provides the designer with a set of tools that allow efficient system development of formally proven models during the design phase.

Atelier B is composed of static analyzers that include (a) the type checker, which is responsible for the syntactic and the semantic verification of a B component, (b) the B0 checker, which performs verification specific to the B0 language[2] [2], and (c) the project verification analyzer, which performs global verifications on the whole of the components of the project, to control its architecture (the links between components).

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:

- The automatic generator of proof obligations from the components in B.
- The base rule manager (the base rule includes more than 2200 rules).
- The automatic prover which releases, without intervention from the user, most of the proof obligations. This prover can be set to a power level to compromise between its speed and proof rate.
- The interactive prover, which is used when the automatic prover has failed. The designer is able to guide the prover with commands and additions of new rules.
- The predicate prover, which demonstrates rules added by the user.
- The graphical visualization of the proof tree of a proof obligation.

Finally, Atelier B supports translation of B implementations to C, C++ and Ada.

5.1. Automatic proof

The automatic proof process in Atelier B requires the following steps [4]:

[1] Atelier B is trademark for ClearSy. It was developed in collaboration with Jean-Raymond Abrial and Alstom Transportation with funding form the RATP, the SNCF and the INRETS.

[2] B0 is a subset of B language, which is used only in implementations to ensure that those can be directly translated in C/C^{++}/Ada.

1. Type Check
2. Proof Obligation (PO) Generation
3. Automatic Proof (with force 'fast', '0', '1', '2', or '3')
4. Interactive Proof (if unproved PO left)

The type checker performs standard type checks that are applied by current compilers usually. Proof obligation generation is producing expressions in predicate logic, which have to be validated by the prover. The proof obligations are generated according to rules given in [2]. Three of these rules, as described in [14], are shown and explained in the sequel. An identifier represents a predicate, except for an identifier in square brackets, which represents a substitution on a predicate.

$$\text{Constraints}_M \Rightarrow \exists v.\text{Inv}_M \quad (1)$$

$$\text{Constraints}_M \Rightarrow [\text{Init}_M]\text{Inv}_M \quad (2)$$

$$\text{Constraints}_M \wedge \text{Inv}_M \wedge \text{Pre}_{op,M} \Rightarrow [\text{Def}_{op,M}]\text{Inv}_M \quad (3)$$

The meaning of each equation is described below:

- **Non-empty state:** For machine M with parameters satisfying their constraints, exists a state satisfying the invariant (Eq. (1)).
- **Initialization:** For machine M with parameters satisfying their constraints, exists an initialization satisfying the invariant (Eq. (2)).
- **Preservation of the invariant:** Each operation is preserving the invariant, if the operation precondition, and the machine constraint and invariant are satisfied (Eq. (3)).

The number of generated proof obligations depends on the size of the invariant, the number of variables, the number of operations and the type and complexity of substitutions in operations. Composing a machine from other machines may increase the number of generated proof obligations considerably, if the invariant of the composed machine incorporates properties of included machines.

After proof obligation generation the automatic prover of Atelier B is used to discharge the proof obligations. The prover uses a rule base, consisting of *rewrite*, *forward*, and *backward* rules. A proof obligation is proven, if it can be transformed to *true* by applying a set of rules. Proof of first order logic predicates as defined for B can be undecidable, therefore the search space for a suitable sequence of transformations is narrowed by heuristics and proof time restrictions in the prover. The 'force' selected by the user affects the 'effort' the prover makes in the process. With increased force more powerful heuristics are applied, with the drawback of increased proof time. Table 1 shows indicative figures on time trade-off with increasing force [4].

Table 1
Automatic prover forces

Force	Approximate proof time	Proof rate (%)
0	always less than 10 s	70
1	from a few seconds to 2 or 3 min	+1
2	from a few minutes to several 10 min	+3
3	from several 10 min to several hours	+1
"fast"	less than 30 s	+30

5.2. Interactive proof

Automatic proof will not necessarily prove every valid proof obligation, as the heuristics built into the prover may not help with particular predicates. However, by inspection of the unproven predicate, a system developer may feel convinced that it is true. To allow proceeding with proof, the proof mechanism in Atelier B offers an interactive 'manual' interface to the prover.

Interactive proof is performed by issuing commands which fall into several categories [4]:

- **moving:** these commands are administrative, e.g. they enable switching between different proof obligations as well as moving within a manual proof by *undo* and *retry* commands.
- **reading**: these commands help in gathering information about the proof obligation.
- **automatic**: these commands call automatic proof algorithms: the standard automatic prover (pr), and the predicate prover (pp).
- **proving**: these commands perform various steps of the proof process, e.g. prove and add hypothesis, deduction, contradiction.
- **rules:** these commands are related to the rules database. The user may search for rules matching a given pattern, or he/she may prove and add his/her own rules to the database.

A good approach in proving a proof obligation is to simplify it to a point where one of the automatic provers is able to perform the proof. The standard automatic prover is just the rule based prover which is invoked when performing an automatic proof. The predicate prover is different in that it does not rely on the rule base. It is based on a couple of axioms and tries to prove a proof obligation by inference. This prover is not used in automatic proof, because of its long run-times. It can be effective in some cases, to invoke the predicate prover when performing interactive proof. Run-time of this prover is restricted to 60 s, but this may be changed by the user.

Interactive proof can take a lot of time, therefore the user should be convinced, that all unproven proof obligations are *true*: if one of the proof obligations in the process turns out as wrong, then the specification has to be adjusted. In most cases this will result in different proof obligations, invalidating the interactive proofs performed so far.

6. Case study: Design and implementation of a telecommunication system based on the HIPERLAN/2 protocol

The main goal of this section is to present the application of the B method/language in a real world case study. The selected application is borrowed from the domain of wireless telecommunication systems and is based on HIPERLAN/2 [6], a standard protocol for broadband wireless networks.

6.1. An overview of the HIPERLAN/2 protocol

HIPERLAN/2 protocol provides data rates of up to 54 Mbits/s for short range (up to 150 m) communications in indoor and outdoor environments. Typical application environments are offices, homes, exhibition halls, airports, train stations and so on.

In order to specify a radio access network that can be used with a variety of core networks, the HIPERLAN/2 standard [12] provides a flexible architecture that defines core independent physical (PHY) and Data Link Control (DLC) layers and a set of convergence layers that facilitate access to various core networks including Ethernet, ATM, and IEEE 1394 (Firewire).

The air interface is based on time division duplex (TDD) and dynamic time division multiple access (TDMA). It relies on cellular networking topology combined with ad hoc networking capability. It supports two basic modes of operation: centralized mode (CM), and direct mode (DM). In the CM operation every radio cell is controlled by an access point covering a certain geographical area and mobile terminals communicate with one another or with the core network through the access point. In the DM operation, mobile terminals in a single cell network can exchange data directly with one another. The access point controls the assignment of radio resources to the mobile terminals. Fig. 3 outlines the protocol architecture, while Fig. 4 delineates the scope of HIPERLAN/2 standards.

Due to the high complexity of the B models of the overall HIPERLAN/2 system, in the next paragraphs part of the overall system design using the B method/language is presented. More specifically, we provide the design details for the Medium Access Control (MAC) Frame Scheduler for the Access Point (AP). The latter, is one of the most complex parts of the HIPERLAN/2 system and constitutes

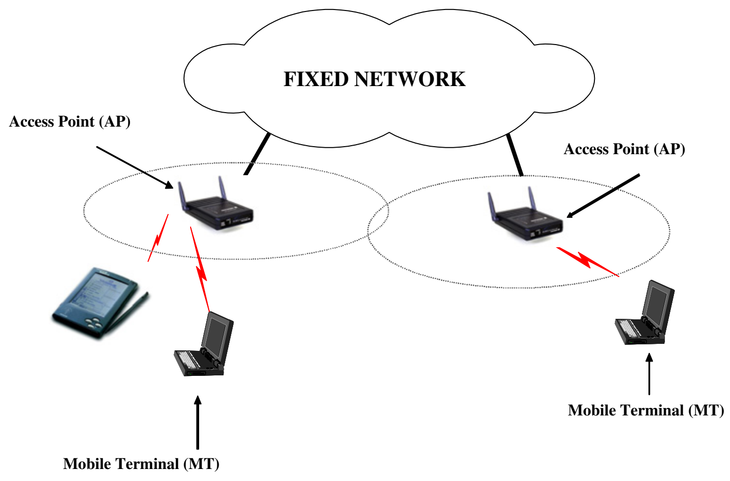

Fig. 3. An overview of HIPERLAN/2 architecture.

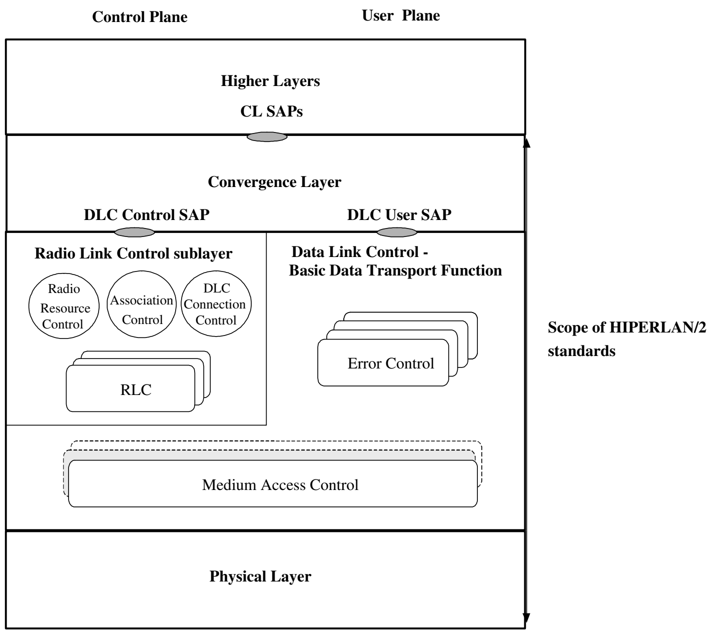

Fig. 4. The scope of HIPERLAN/2 standards.

about 20% of the overall system. It lies in the MAC sublayer of the DLC layer and is responsible for the design of MAC frames. The structure of a MAC frame is illustrated in Fig. 5 [7]. The BCCH, FCCH, RFCH, RBCH, UBCH, UNCH, UDCH, DCCH, LCCH and ASCH are the logical channels of a MAC frame and are defined for different kinds of data transfer services as offered by the MAC entity. The logical channels are mapped onto different transport channels, which are the basic elements to construct Protocol Data Units (PDU) trains. The BCH, FCH, ACH, RCH, LCH and SCH are the transport channels. The LCH and SCH channels are included in the DL-phase and UL-phase of Fig. 5. Fig. 5 depicts also the mapping between logical and transport channels. A logical channel is mapped to the transport channel with the same colour (the BCCH is mapped to BCH, FCCH to FCH, RFCH to ACH, ASCH to RCH, RBCH to SCH and LCH, UBCH to LCH, UMCH to LCH, UDCH to LCH. DCCH is mapped to SCH and LCH for the downlink and the direct phases, while it is mapped to SCH, LCH and RCH for the uplink (the line from DCCH to RCH in Fig. 5). LCCH is mapped to SCH for the downlink and the direct phases, while it is mapped to SCH and RCH for the uplink (the line from LCCH to RCH in Fig. 5.)). For more information about the functionality of the logical and transport channels please refer to [7].

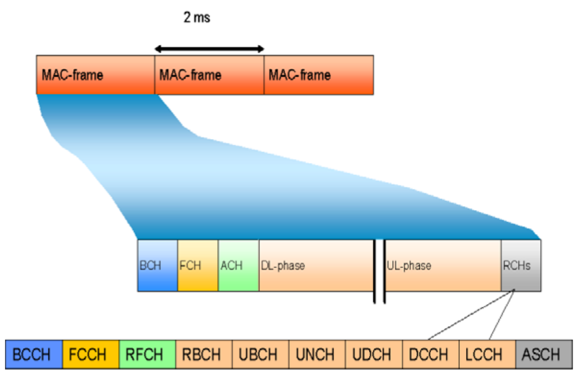

Fig. 5. The structure of a MAC frame.

6.2. System analysis and specification

When the system powers up, the subsystem implementing the system scheduler is initialized. From that point on, the scheduler is triggered by the hardware every time a MAC frame has to be emitted. The scheduler constructs the new MAC frame according to the resource requests by mobile terminals and the limitations of the system specifications. First of all, a table summarizing the traffic flow in the system is updated properly, taking into account the requests by mobile terminals and then

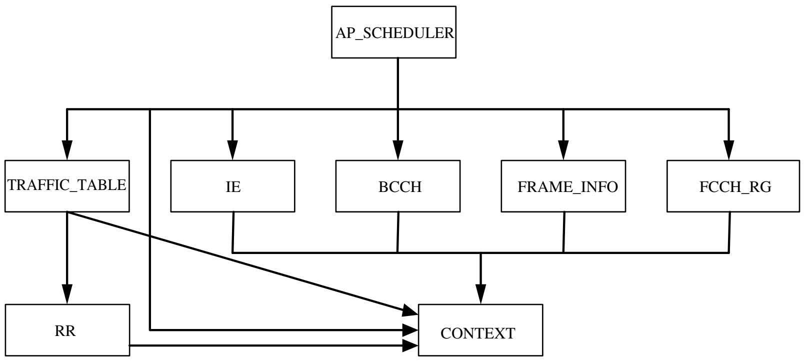

Fig. 6. The decomposition model of the initial B modules.

the appropriate logical – transport channels are filled. Finally, a channel describing the whole structure of the frame (frame channel) is constructed. As soon as a new frame is designed, the frame builder is triggered to construct and transmit it properly.

The first step was to select the initial modules. Some of these modules need to communicate, which means that a module "needs" another module. So we had to define a hierarchical organization for these modules constituting the AP frame scheduler. The AP Scheduler serves user requests set on mobile terminals, so we need a module containing the resource requests by mobile terminals and another module summarizing the traffic flow in the system. All the other modules needed are used to describe the contents of the logical and transport channels. The resulted decomposition is illustrated in Fig. 6. The boxes in Fig. 6 represent discrete B modules and the arrows introduce relationships between them (an arrow sets a "needs" relationship between two B modules).

The functionality of the modules of the organization in Fig. 6 is as following:

- AP_SCHEDULER: responsible for the design of a MAC frame. Specifically, it calculates the final number of LCH and SCH channels[3] that will be granted and creates the contents of the FCH channel. Any other activity required it is performed by appropriate calls of the scheduler.
- TRAFFIC_TABLE: describes the next frame's logical channel entries required (traffic flow), according to the resource requests.
- IE: describes the contents of the information elements (IEs) for the downlink and uplink phases, the idle parts (idle IEs) and the padding IEs of the frame. These IEs will be used in the creation of the FCH channel.
- BCCH: contains the contents of the BCCH logical channel.
- FRAME_INFO: decides the number of IEs, the number of blocks a block contains three IEs [7], the number of idle IEs, and the number of padding IEs.
- FCCH_RG: contains the resource grants for the FCCH channel.
- RR: not a part of the scheduler, but it contains the resource requests imposed by the mobile terminals.
- CONTEXT: a header file for the application and contains only commonly used definitions (it is visible by all modules).

The next step was to create an abstract model for each module. Following the rules of the B method/language, a set of B abstract machines was created to specify the abstract modules required. Within each abstract machine, the main subsystem properties were formally described. Every machine was fully proven to be correct with the use of Atelier B's automatic and interactive provers [1]. Appendix B provides the details of the FRAME_INFO abstract machine, as a simple example of the system under development.

6.3. System design using the B method/language

In system design phase, the abstract machines of each subsystem were formally refined to B

[3] LCH and SCH are transport channels that constitute the DL- and UL-phases (illustrated in Fig. 5), respectively.

refinements or implementations (the last level of refinement). Appropriate predicates were defined to express the properties of the linking (gluing) invariant between each B refinement/implementation and its corresponding abstract model. The proof obligations generated were in most cases proven using the automatic prover of Atelier B. Nevertheless, there were also cases where the designers had to prove several proof obligations interactively.

Due to the complexity of the proof obligations generated by B implementations, in several cases the designers experienced excessive numbers of proof obligations (sometimes more than 100), which were impossible to be proved using the interactive prover. In those cases, there were two different approaches to follow:

- Introduction of an intermediate refinement level (see Fig. 1) between the abstract model and the corresponding refinement/implementation, to express some properties at an intermediate level of detail.
- Introduction of a new module (or more if necessary) to simplify the proving procedure. The new module can simplify the complex operations (the ones that create excessive numbers of proof obligations). The machine of the new module is imported to the implementation with the use of the IMPORTS clause (e.g. the modules IE_SINGLE and BIT_MANIPULATION were used to reduce the number of proof obligations generated for the IE implementation component).

First, intermediate refinements were tried to be introduced during the development of the project. The result was that they had been generated some proof obligations that it was very hard to prove even with the interactive prover. So, the whole strategy was changed, trying to introduce some new modules, which really simplified the proving task. After all, in the final model mainly decomposition has been used.

The final decomposition of the initial model is presented in Fig. 7. The dashed arrows represent a SEES clause, while each solid arrow refers to an

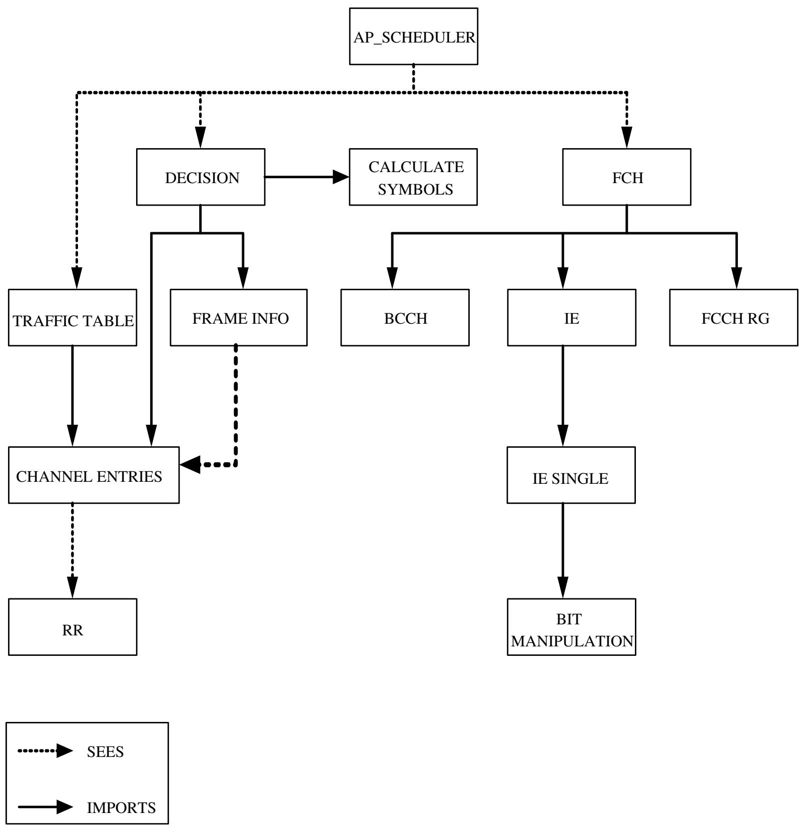

Fig. 7. The final decomposition model.

IMPORTS clause. When a component SEES an abstract machine M, the data of M may be accessed as read-only, and the modification operations of M can not be called. The IMPORTS clause may only appear in implementations, and imports one or more abstract machines in order to implement data and operations within lower level machines. For more information about the SEES and IMPORTS clauses please refer to Appendix A.

The newly introduced B modules have the following functionality:

- BIT_MANIPULATION: stores a value in an octet of a buffer.
- IE_SINGLE: stores the proper values to the fields of the IE buffer structure separately [7].
- CHANNEL_ENTRIES: stores the contents of the proper channel entries of the frame separately according to the resource requests.
- DECISION: calculates the rates of LCH, SCH, control and data channels that will be finally used according to the resource requests.
- CALCULATE_SYMBOLS: calculates the number of symbols for a single transmission according to the physical mode that will be used.
- FCH: creates the FCH channel, which describes the contents of the frame.

7. Evaluation of the B method

In order to evaluate the proposed method, the B design/implementation of the HIPERLAN/2 protocol has been compared to an existing implementation of the same protocol parts. For both implementations, the specifications provided by ETSI for HIPERLAN/2 were used [7].

The HIPERLAN/2 Access Point scheduler has been already developed in UML 1.4 [18] and is available in executable form. UML was used in every design phase, and the validation of the final system was accomplished through simulation of C++ code. The same team employed in the original development of the scheduler, has also been working for the development of the scheduler using the B method/language. The actual goal of the latter was to implement the same part of the protocol in C++, and compare the two design alternatives.

As described in Table 2, the person months spent in system specification phase are slightly increased in the case where the B method/language was used. This is due to the fact that the definition of the abstract machines of each subsystem at the specification stage included definitions of the appropriate invariants. Additionally, the time spent for model refinement is also increased compared to the ad hoc model refinement, taking place in UML based design. The time increase in the case of the B method/language is because the implementation of each abstract machine in B must be proven compliant with the abstract machine's invariants. The latter must hold at every step of the final implementation described in B. As far as validation is concerned, no time was spent in the B based approach since the compliance of the final implementation with the initial specification has already been proven formally during refinement. On the contrary, in the UML based design 1.4 person months have been spent for validation (in this case validation was achieved through code generation and simulation of the code produced). Finally, in both case studies similar amount of time spent for integrating the code of the AP scheduler in the HIPERLAN/2 prototype.

Regarding the overall time spent in the case of the B method/language, it is slightly decreased compared to the time spent in the case of the UML based design. The main reason for this was the fact that the designers involved in the development of the B models were not familiar with the use of formal methods, and especially with the efficient definition of *invariants* and *preconditions*. In several cases, it was necessary to redefine them in order to reduce the number of proof obligations generated. This is actually the reason why in Table 1, the specification and model refinement phases appear to be prolonged in the case of the B method/language. Moreover, the people participated in the design of the system, were already familiar with UML based design, as result they spent less time during the specification and model refinement phases. In both

Table 2
Person month allocation per design phase

AP Scheduler	Specification	Model refinement	Validation	Integration in prototype	Total
B based design	1.1	2.2	–	0.35	3.65
UML based design	0.75	1.3	1.4	0.25	3.7

cases, the designers were well aware of the intrinsic details regarding the HIPERLAN/2 protocol.

As a result, it is expected that the time spent for designing the system using the B method/language will be significantly reduced in the case where the same design team will develop a similar system.

Table 3 delineates the time spent for writing and proving the specifications and refinements from B abstract machines to implementations for each of the components resulting from the initial system specification. The purpose of the decomposition task is to enhance the proving process. So, all the proof obligations had to be discharged were generated by the invariants and the preconditions of the various system modules, and the gluing invariant connecting each module with the corresponding module of the previous refinement level. As it can be seen in Table 3, sometimes there is a great variation between the numbers of proof obligations of different modules. This is due to the complexity of the module, the number of operations, the loops inside the operations, the preconditions and the invariants included. Some of them were too simple and had no proof obligations (e.g. the CONTEXT machine (context.mch) and implementation (context_1.imp) contain only declarations of global variables and so there is nothing to prove). One can apply one of the approaches described in Section 6.3 (e.g. introduction of a new module) to keep the number of proof obligations more reasonable, but this is not always applicable. Moreover, some modules produced an increased number of proof obligations. Most of them were proven automatically, while the remaining proof obligations were proven interactively (this is the case of the module ie_single).

In the specific case study, there were cases where the designers had to address a number of proof obligations that could not be discharged. The latter emerged due to one of the following reasons:

- errors in the specification (which were not formally specified),
- missing predicates describing properties of variables in the specification or in the implementation's gluing invariant.

Concerning the first case, the use of the B method/language revealed five errors in the initial system specification of the AP scheduler. The errors identified during the first refinement of the B abstract machines describing the AP scheduler. Throughout the refinement process, additional errors were also identified due to missing predicates describing variable properties. In order to deal with them, the designers had to add additional predicates describing the missing properties in the specification or in the implementation's invariant to make the proving task easier. Sometimes extra components were added (as in the IE or the TRAFFIC_TABLE case), which were imported to the implementation of a module (the IE_SINGLE, BIT_MANIPULATION and CHANNEL_ENTRIES machines) to restrict the number of non proven POs.

Regarding the use of Atelier B for supporting the B method/language, there are improvements that could make the tool even more flexible and efficient during the proving process. For example, the rule base of the tool could be enriched in order to be able to prove more complex proof obligations. In this way, a significant 'burden' regarding the proper use of the B language will move form the designers

Table 3
Proof obligations required for the AP scheduler case study

Component	POs			Time spent (PH)
	Generated	Autom. proven	Inter. proven	
context.mch	0	0	0	1
context_1.imp	0	0	0	1
bcch.mch	8	8	0	1
bcch_1.imp	17	17	0	2
bit_manipulation.mch	1	0	1	6
bit_manipulation_1.imp	31	31	0	9
calculate_symbols.mch	0	0	0	1
calculate_symbols_1.imp	12	12	0	1
channel_entries.mch	152	152	0	20
channel_entries_1.imp	444	414	30	49
decision.mch	2	2	0	8
decision_1.imp	615	571	44	138
fcch_rg.mch	17	17	0	1
fcch_rg_1.imp	55	47	8	2
fch.mch	5	5	0	9
fch_1.imp	715	655	60	112
frame_info_rg.mch	0	0	0	10
frame_info_rg_1.imp	159	156	3	19
ie_single.mch	36	36	0	16
ie_single_1.imp	649	600	49	30
ie.mch	32	32	0	6
ie_1.imp	24	19	5	21
rr.mch	0	0	0	24
rr_1.imp	21	12	9	23
ap_scheduler.mch	0	0	0	8
ap_scheduler_1.mch	5	5	0	10
traffic_table.mch	1	1	0	13
traffic_table_1.imp	342	314	38	43
Total	3343	3106	247	584

to the tool. This is considered one of the main success factors of the use of the B method/language in industry.

8. Conclusions

The previous sections presented the use of the B method/language for the design of complex telecom applications. As opposed to existing design practices of telecom applications, the one presented in this paper exhibited the use of the B method/language for building abstract system models, and describing their properties in a formal way. The final implementation emerged as a result of successive refinements, which were formally proven to maintain the properties of the initial system specification. This is in contrast to ad hoc refinement supported by most design methods. The proof obligations generated during formal model refinement were proven using the Atelier B tool.

The proposed approach has been applied in practice through the design of a case study where part of a real world telecom system, based on HIPERLAN/2 protocol, has been designed and implemented. An overall evaluation has also been presented in order to identify the benefits of using the B method/language, while in parallel to keep track of the potential inefficiencies related to it. The B method/language appears to be a promising approach for the design of complex telecommunication systems since it allows correct-by-construction implementations. Formal proof of system properties during model refinement guarantees less time spent in verification and validation phases, thus leading to shorter development cycles.

The approach presented in the previous sections has been adopted for the design of critical protocol parts that are common in a variety of telecom products. For that purpose, a library of formally proven telecom components has been developed. Each component has been specified in the B language, and has been appropriately refined so as produce error free component instantiations in C++. The component descriptions, which form a valuable starting point for system level design of telecom products, they are solely described in B.

Based on the concepts presented in the paper, the B method/language appears to be an approach that can deal with the ever increasing complexity of telecom systems, since it provides the ability to reduce the time required for system validation, which constitutes one of the main bottlenecks in the design of complex telecom systems. In an industrial environment, the B method/language could be a promising alternative, though there are improvements that have to be done in order to make the language widely accepted. One of the main obstacles for adopting the language in a product line is that it requires extensive training and strong background in formal methods in order to use it effectively. In the context of the HIPERLAN/2 case study, the design team consisted of designers with such back ground. The real challenge is to improve both the language and the supporting tools so as to be more comprehensive and more intuitive. In that direction, it would be beneficial to combine the B language/method with other specification languages e.g. UML [18] and SDL [20], that are heavily used for the specification of telecommunication systems.

Appendix A. The B method/ language

The B method/language is a specification method based on a mathematical formalism with proofs guaranteeing that implementations conform to their specifications. The B method/language is suitable for implementing both hardware and software and it is currently used in the industry for the development of control systems software, the validation of specifications and the validation/re-engineering of critical software. It is also used in the context of real-time systems, industrial automation, communication protocols and management information systems with complex data models, characterizing a system in terms of its properties. This means that the B method/language guarantees that the specified properties will be non-ambiguous, mutually consistent and respected by the final implementations.

The heart of the B method/language is the concept of a module. A B module corresponds to a subsystem model. A fully developed B module consists of several B components: an abstract machine (the module specification), a number of possible refinements and an implementation (the final refinement containing the B0 code, the B code that can be executed). These components are maintained within a single B project. Each B component contains:

- a static aspect including the definition of the subsystem state space (sets, constants, variables) and the invariant, which is the definition of static properties for its state variables.
- a dynamic aspect including the definition of the initialization phase for the state variables and the definition of operations for querying or modifying the state.

An abstract machine is the formal specification of a software module. It defines a mathematical model of the subsystem concerned containing an abstract description of its state space and possible initial states and an abstract description of operations to query or modify the state. A typical form of an abstract machine contains the following parts:

```
MACHINE (The name of the abstract
machine)
SETS (The names of the sets used as
definition domains for variables and constants)
CONSTANTS (The constant names)
PROPERTIES (Predicates describing the
properties of constants only)
VARIABLES (The variable names)
INVARIANT (Predicates describing the
properties of variables only)
INITIALISATION (A set of substitutions)
OPERATIONS (A set of operation
definitions)
END
```

The INITIALISATION and OPERATIONS parts constitute the dynamic aspect, while all the remaining parts constitute the static aspect of an abstract machine. To precise that the operation content is defined only in some conditions, some logical predicates are used, known as preconditions. A precondition points out that the operation has to be applied only when the condition holds. If an operation has input parameters, a minimal precondition giving sense to the operation is to type these parameters.

The refinements are components that refine an abstract machine (or its most recent refinement). They add new properties to the previous mathematical model to make it more concrete. This is accomplished with the introduction of new variables to represent the state variables for the refined component, plus their linking (or gluing) invariant, which is a set of predicates describing the relationships of the variables of the abstract machine (or its recent refinement) with the variables of the corresponding refinement. A general form of an intermediate refinement contains the following parts:

```
REFINEMENT (the name of the refined
machine extended by "_n", where n is the
refinement level)
REFINES (the name of the refined machine)
VARIABLES (the names of the
new variables)
INVARIANT (properties of the new
variables and the linking invariant)
INITIALISATION (initialization refinement)
OPERATIONS (operation refinements)
END
```

The last level of refinement is called implementation and contains the executable B code, called B0.

A B project is a set of linked B modules. The principal dependencies links between modules are the IMPORTS links and the SEES links. The IMPORTS clause is used to form a modular decomposition tree and it may only appear in implementations. An implementation imports other abstract machines in order to implement data and operations with lower level machines. The variables of an imported machine may be modified in the implementation by operation calls. When a component SEES an abstract machine, the data of the abstract machine may be accessed as read-only and the modification operations of the abstract machine can not be called. The SEES clause is not transitive.

Appendix B. An example of B code

The next paragraphs provide an insight on the abstract machine of *frame_info* block of the Access Point scheduler of HIPERLAN/2 protocol, and its corresponding implementation in B. The frame_info block is used to compute the length of the FCCH channel. The FCCH is sent in downlink direction and conveys information that describes the structure of the MAC frame visible at the air interface. The following information can be transmitted in the FCCH [ETSI. 2000]:

1. A resource grant for SCHs and LCHs that can be used to transport information for RBCH, DCCH, LCCH, UDCH, UBCH and UMCH.

2. Announcements of empty parts of the MAC frame.

FCCH is carried by FCH.

The operations defined in the frame_info abstract machine are the estimation of the number of Information Elements (IEs), Information Blocks (IBs), empty parts (idle IEs) and padding IEs. Every IB contains three IEs. In case that there is a power measurement request, one more IB is added if there are not padding IEs. If the remainder of the number of connections requested divided by 3 is not 0, then we complete the last block with padding IEs (if there is a power measurement request too, we substitute one padding IE with an empty one). Padding IEs are used to fill an FCH IE block if the total number of resource grants does not end up in a multiple of three. The total number of IEs is always kept as a multiple of three, because every block can contain three IEs [7].

In the following code, the TrafficTableSize is a constant contained in the context machine, equal to 62, calculated as following: we consider a maximum number of ten Mobile Terminals per Access Point that can be supported simultaneously and three DLC connections for each one of them. Every DLC connection consists of a downlink and an uplink phase. Then we have three connections for the downlink and three connections for the uplink phase for each mobile terminal. We have also considered one more connection for the BCH and one for the RCH channels. Consequently, we can deduce that TraficTableSize = 10*(3 + 3) + 2 = 62.

```
MACHINE
  frame_info
SEES
  context, channel_entries
CONCRETE_VARIABLES
  ib_number, /*the total number of
  Information Blocks */
  ie_idle, /* the number of empty IEs
  */
  ie_number,/* the number of IEs
  requested (equals the number of
  DLC connections requested) */
  ie_pad /* the number of padding IEs
  */
INVARIANT
  /* the number of DLC connections
  requested divided by 3, plus one
  extra block added if there is a
  remainder in the division, or there
  is a need for an extra block as
  described in the header comments
  of the machine */
  ib_number ∈ 1...(TrafficTableSize/
  3)+1∧
  ie_idle ∈ 0...2∧
  /* the max number of IEs requested
  equals the max number of DLC connec-
  tions requested*/
  ie_number ∈ 0...TrafficTableSize∧
  ie_pad ∈ 0...2 ∧
  /*all the empty and padding IEs are
  contained in the same block */
  not(ie_idle = 2∧ ie_pad = 2)∧
  /* every block contains 3 IEs */
  ie_idle + ie_pad ⩽ 3∧
  /* the total number of IEs should be
  a multiple of 3 */
  ie_num-
  ber + ie_idle + ie_pad = 3*ib_number
  ∧
  /* if there are no connections
  requested we create one block
  filled with empty and padding IEs */
  (ie_number = 0 ⇒ ib_number = ie_
  number+1)∧
  /* Equality holds when there is only
  one connection requested */
  (ie_number ⩾ 1 ⇒ ib_number ⩽ ie_
  number)∧
  /*the upper bound for the total num-
  ber of IEs */
  ie _number + ie_ idle + ie_ pad ⩽
  TrafficTableSize + 1
INITIALISATION
  ib_number: = 1 ||
  ie_number: = 0 ||
  ie_idle: = 0 ||
  ie_pad: = 0

OPERATIONS
  estimate_fch_length (power_m_r)=
  PRE
    power_m_r ∈ BOOL
  THEN
    ANY l_ie_number, l_ie_idle WHERE
    l_ie_number = traf_table_index&
    l_ie_idle ∈ 0...2&
    /*if there are no connections
  requested, one block is created con-
  taining empty and padding IEs */
  (traf_ table_index  = 0 ∧ power_m_
  r = TRUE ⇒ l_ie_idle = 2)∧
  (traf_  table_index   = 0 ∧ power_
  m_r = FALSE ⇒ l_ie_idle = 1)∧
  (traf_table_index > 0 ∧ power_
  m_r = TRUE ⇒ l_ie_idle = 1)∧
  (traf_table_index > 0 ∧ power_
  m_r = FALSE ⇒ l_ie_idle = 0)
THEN
  /* the IEs requested are always
equal to the number of connections
*/
  ie_number: = l_ie_number ||
  ie_idle: = l_ie_idle ||
  ie_pad, ib_number: (
    ie_pad ∈ 0...2∧
    ib_number∈   1...(TrafficTable-
Size/3)+1∧
    not(ie_idle = 2 ∧ ie_pad = 2)∧
    ie_idle+ie_pad ⩽ 3∧
    (l_ie_number = 0 ⇒ ib_num-
ber = l_ie_number + 1) ∧
    (l_ie_number
⩾ 1 ⇒ ib_number ⩽ l_ie_number)∧
    3*ib_number
= l_ie_number + l_ie_idle + ie_pad)∧
    ie_number + ie_idle + ie_pad ⩽
TrafficTableSize + 1
  END
END
END
```

IMPLEMENTATION
```
  frame_info_1
```
REFINES
```
  frame_info
```
SEES
```
  context, channel_entries
```
INITIALISATION
```
  ib_number: = 1;
  ie_number: = 0;
  ie_idle: = 0;
  ie_pad: = 0
```
OPERATIONS
```
  estimate_fch_length(power_m_r)=
  BEGIN
    IF traf_table_index = 0 THEN
  /* Create one block with padding and
  empty IEs */
    ib_number: = 1;
    ie_number: = 0;
    ie_idle: = 1;
    ie_pad: = 2
    ELSE
  /* the IEs requested are always
  equal to the number of connections
  */
    ie_number: = traf_table_index;
    ie_idle: = 0;
    IF traf_table_index > 2 THEN
    /* every block contains 3 IEs */
    ib_number: = traf_table_index / 3;
    VAR xx IN
    xx: = ib_number*3;
    IF xx ≠ traf_table_index THEN
  /* if there is a remainder in the
  division fill in the last block with
  padding IEs */
      ib_number: = ib_number + 1;
      ie_pad: = ib_number*3-
  traf_table_index
    ELSE
  /* if there is not a remainder in the
  division there are not padding IEs
  */
      ie_pad: = 0
    END
  END
  ELSE
      /* If only one IE is requested
  create one block */
      ib_number: = 1;
      ie_pad: = 3-traf_table_index
    END
  END;
  IF power_m_r = TRUE ∧ ie_pad > 0 THEN
  /* Substitute one padding by one
  empty IE */
    ie_pad: = ie_pad-1;
    ie_idle: = ie_idle+1
  ELSIF  power_m_r = TRUE ∧ ie_pad = 0
  THEN
  /* Create an extra block with empty
  and padding IEs */
    ib_number: = ib_number + 1;
    ie_idle: = ie_idle + 1;
    ie_pad: = 2
    ELSE skip
  END
  END
  END
```

References

[1] B. Atelier http://www.AtelierB.societe.com/, 2005.

[2] J-R. Abrial, The B Book: Assigning programs to meanings, Cambridge University Press, 1996.
[3] J. Bowen, Specification and Documentation using Z: A Case Study Approach, International Thomson Computer Press, 1996.
[4] Clearsy, Interactive Prover User Manual, Version 3.2.2, 2003.
[5] J. Draper et al., Evaluating the B-method on an avionics example, in: Proceedings of Data Systems in Aerospace (DASIA) Conference, Rome, Italy, European Space Agency Publication Division WPP-116, 1996, pp. 89–97.
[6] ETSI http://www.etsi.org/, 2005.
[7] ETSI, Broadband Radio Access Networks BRAN; HIPERLAN Type 2; Data Link Control (DLC) Layer Part1: Basic Data Transport Functions. ETSI TS 101 761-1 v1.1.1, 2000.
[8] U. Glasser, Systems Level Specification and Modeling of Reactive Systems: Concepts, Methods and Tools, in: Proceedings of EUROCAST 95, Springer Verlag, 1995.
[9] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
[10] IEEE, Software Engineering Standards, The Institute of Electrical and Electronics Engineers, 1987.
[11] C.B. Jones, Systematic Software Development using VDM, Prentice Hall International, 1990.
[12] J. Khun-Jush, et al. 2002. HIPERLAN Type 2 for Broadband Wireless Communication. Ericsson Review No.2.
[13] T. Kroph, Introduction to Formal Hardware Verification, Springer Verlag, 1998.
[14] A Krupps, W. Mueller, D4.3.2: Refinement and verification of real-time properties. IST-2000-30103 PUSSEE, Project Report. Available at: http://www.keesda.com/pussee, 2003.
[15] L. Lamport, The Temporal Logic of Actions, ACM Transactions on Programming Languages and Systems 16 (3) (1994).
[16] G.J. Milne, Circal and the Representation of Communication, Concurrency and Time, ACM Transactions on Programming Languages and Systems 7 (2) (1985).
[17] R. Milner, Communication and Concurrency, Prentice Hall, 1989.
[18] OMG, http://www.omg.org/, 2005.
[19] C. Snook, L. Tsiopoulos, M. Walden, 2003. A Case Study in Requirement Analysis of Control Systems using UML and B, in: Proceedings of International Workshop on Refinement of Critical Systems:Methods, Tools and Developments, Turku, Finland.
[20] L. Doldi, Validation of Telecom Systems with SDL, Wiley, 2003.

Konstantinos Antonis was born in Lamia, Greece, in 1971. He graduated from the department of Computer Engineering and Informatics at University of Patras in Greece in 1994, and he received his PhD diploma from the same department in 2000. His main research interests include distributed systems, formal methods and distance education. In the past, he has been working for Intracom Telecom Solutions S.A. Now, he is a lecturer at the department of Informatics and Computer Technology at Technological Educational Institute of Lamia (Greece).

Nikolaos S. Voros received his Diploma in Computer Engineering and Informatics Engineering, in 1996, and his PhD degree, in 2001, from University of Patras, Greece. His research interests fall in the area of embedded system design, and include specification techniques for complex embedded telecommunication systems, hardware-software co-design, formal methods, pattern based design and reconfigurable systems. During the last years he has been working as consultant and technology specialist in several organizations and institutes including the Department of Computer and Electrical Engineering (University of Patras, Greece), Industrial Systems Institute (Greece), Computer and Technology Institute (Greece) and Intracom Telecom Solutions S.A. Dr Voros is currently Assistant Professor at Technological Educational Institute of Mesolonghi, Department of Telecommunication Systems and Networks. He has published more than 40 referred articles in international journals and conferences. He is also co-author of the books *System Level Design Model with re-Use of System IP and System Level Design of Reconfigurable Systems-on-Chip* published by Springer.