HAL
archives-ouvertes.fr

Extended Model driven Architecture to B Method

Ammar Aljer, Philippe Devienne

▶ To cite this version:

Ammar Aljer, Philippe Devienne. Extended Model driven Architecture to B Method. Ubiquitous Computing and Communication Journal, UBICC publishers, 2011, Special Issue on ICIT 2011. hal-00832612

HAL Id: hal-00832612

https://hal.archives-ouvertes.fr/hal-00832612

Submitted on 11 Jun 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

EXTENDED MODEL DRIVEN ARCHITECTURE TO B METHOD

Ammar Aljer
Faculty of Electrical and Electronic Engineering, University of Aleppo, Aleppo, Syria
ammar.aljer@lifl.fr

Philippe Devienne
Lille's Computer Science Laboratory, University of Lille, Lille, France
philippe.devienne@lifl.fr

ABSTRACT

Model Driven Architecture (MDA) design approach proposes to separate design into two stages: implementation independent stage then an implementation-dependent one. This improves the reusability, the reliability, the standability, the maintainability, etc.

Here we show how MDA can be augmented using a formal refinement approach: B method. Doing so enables to gradually refine the development from the abstract specification to the executing implementation through many controled steps. Each refinement step is mathematicaly represented and is proven to be correct, by conceconce then the implementention is proven to satisfy the specification; furthermore this approach permits to prove the coherence between components in low levels even if they are branched in different technologies during the development.

Keywords: MDA, B method, Co-design Refinement, Embedded System, VHDL

1 INTRODUCTION

As computer performance improves and human-built systems augment, there are continuous efforts to employ suitable Computer Aided Design tools that are able to develop such complex systems. A common attitude between designers in different technologies is to use more abstract design levels that enable designer to concentrate, at first, on the most important requirements of the system.

In hardware domain, many tools are produced to develop higher levels than printed circuits or RTL (Register Transfer Level). VHDL (IEEE 1076) is emerged on 1987. It permits to represent a complete hardware system. It became the dominant in Hardware modelling. VerilogSystem is standardised in 2005 to manage abstract level of hardware system. In software area, number of OOP languages has emerged. They give more facilities to treat complex system than procedural languages. An implementation-independent tool, UML (unified modelling language), use graphical diagrams to gather common aspects of OOP Languages using. An object oriented system is made up of interaction components. Each component (object) has its own local state and provides operations on that state. In Object oriented design process, Designer concentrates more on precising classes (abstraction of real objects) and the relationships between these classes. MDA (model driven architecture) was launched by the OMG (Object Management Group) in 2001. It proposes to separate the design into two stages: implementation-independent stage then an implementation-dependent one. "The transition between these stages of development should, ideally, be seamless, with compatible notation used at each stage. Moving to the next stage involves refining the previous stage by adding details to exiting object classes and devising new classes to provide additional functionality. As information is concealed within objects, detailed design decision about the representation of data can be delayed until the system is implemented."[8].

Another important aspect of nowadays systems is the interference between different technologies. Most systems consist of different cooperating sub-systems where some functionality may migrate from one technology to another in further versions of the system.

In our project, which is illustrated in Fig. 1, we improved MDA approach in three main aspects:

1. Smoothing transfer from the abstract specification of the system into the implementation with a proven refinement from each level to the next and the more deterministic one.
2. Formal notation of the complete system in the abstract levels

3. Formal projection of components that are implemented in hardware technology.

Our approach (that joints the advantages of MDA and B method) permits to obtain many advantages:

1. The possibility to obtain a correct-by-design system
2. Increase the reusability: when a modification is necessary, we preserve all design levels that are more abstract than the level where modification is occurred.
3. The possibility of migration between technologies in low levels without reproving the complete system if the immigration preserves the logical behaviour captured in the formal projection.

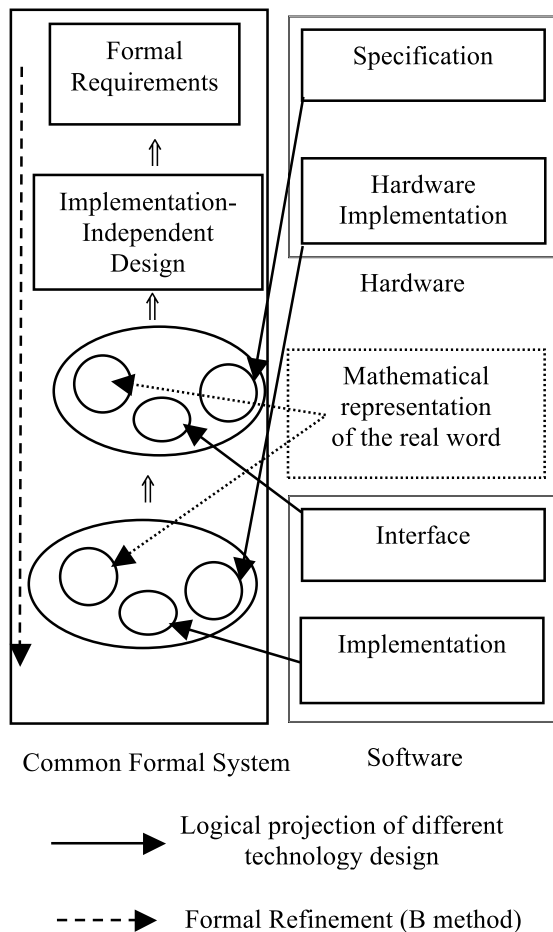


Figure 1: Refined MDA

The dashed line in Fig. 1 shows the temporal axe of project development. The top of the left part of Fig. 1 shows that the first step is to formally specify the requirements. This stage may be achieved during an iterative process where new requirements do not contradict with the previous ones. The Formal requirements specification is followed by another stage to design the main components of the wanted system independently of the implementation technology. Also this stage is, in most cases, achieved iterative process during many steps of refinement. In real applications the previous two stages (formal requirements and the implementation independent design) are not completely separated. Using the formal refinement of B, components in each step is proven to be coherent and refine the previous step. Right part of Fig. 1 shows how designers in each community may their own development tools and techniques to partially implement the system. A formal representation of the implementation of the different technologies is traced to prove:

1. the correctness of each component regarding to its specification,
2. the coherence between components in low levels either if they are implemented in one technology or different technologies.
3. the satisfaction of the Implantation-Independent Architecture declared in the previous stage.
4. and the coexisting, if necessary, with mathematical representation of parts of the real environments such as physical laws, external systems, etc.

2 MODEL DRIVEN ARCHITECTURE

Since the invention of Newmann, the general attitude of software tools developer is to abstract Newmann computer architecture.

FORTRAN may be considered as the first high level language. From the outside, it uses formal mathematical-like expressions but actually these expressions and instructions are chosen to abstract the executive machine code. A compiler is written to convert each FORTRAN program code into machine code. Programs were used to partially help client with automatically and rapidly executing an algorithm. Most of later software developments (such as structural programming then OOP) concentrated on the abstraction of the executive machine code. With OOP, programmer concentrate more and more on the Classes that are abstractions of real word. Nowadays writing the implementation is partially automated and designer may give more attention on system structure. Actually with CASE (Computer Aided Software Engineering) tools, programmer can graphically specify the components of his/her design, precise the operation of each component and defines the relations between components then executive code is automatically generated. Nowadays computer is used not only to execute a program but to represent a complete system and furthermore to simulate a complex of interacting systems. With MDA (Model Driven Architecture) design is completely separated between implementation-independent stage and an implementation-dependent one. With this attitude to

represent as system rather than a program, verification becomes more and more difficult because its cost increases exponentially with complexity. With such approach, Reusing is augmented.. In OOP, programmer reuses ancient classes or libraries (written by him or by others) in new projects. With COSTS (Commercial, off-the-shelf), programmer reuses a complete software system or sub-system. He ought to adapt them to the novel environment.

Since 1950s, huge efforts are made to cover microinstructions with many abstraction layers: Assembly, High Level Languages, Structural Programming, OOP, UML (Unified Modeling language) and MDA. But only few efforts are made to formulate the other side of the programming task; that is client requirements. With the increasing machine power and augmenting complexity of computer based systems, Software engineering developed many principles and techniques to formulate client requirements. Comparing to the development of programming language, theses efforts rest primitive and a formal gap between what a program do and what a client wants is always exists.

SDLC (System Development Life Cycle) in Software engineering usually begins with requirement specification [10] and many UML diagrams partially describe requirements such as Use Case Diagrams, Activity Diagrams .etc. These representations of requirements are still superficial, non formal (or semi formal) and no formal linkage is defined to link these requirements with the corresponding implementation code.

3 HDL, HARDWARE DESCRIPTION LANGUAGES:

Due to the difference between hardware product and software product, Production of hardware or software component passes through tow different sequences. Software engineers concentrate on requirement collection, development, verification, deployment .etc. Hardware engineers emphasis on functional level, logic gate level, RTL (Register Transfer Level) and printed circuit level. The increasing system complexity obligates both communities to develop their tools towards abstract system level.

3.1 VHDL

VHDL that is the dominant language in hardware design was the first to take system level in account.

Even if VHDL [2] was designed for electronic design automation to describe VLSI circuits, it argues that it can be used as a general-purpose language and even can handle parallelism. From hardware community point of view, VHDL may be used to describe the structure of the system since any circuit may be defined as a black box (ENTITY) where all the inputs and outputs are defined then by a white box (ARCHITECTURE) where all the components and connections between these components are declared. Components in the architecture are functionally defined and they could be mapped later to the real word components by an additional level (CONFIGURATION). So it is supported with libraries that contain all specifications of electronic units known in the world. These layers permit to simulate the real circuit in order to verify the design. ARCHITECTURE layer in VHDL may define the behaviour of the circuit instead of its structure. Beside VHDL most important HDLs , such as SystemVerilog and SystemC respect the distinction between abstract and implemented levels.

3.2 HDL and Co-Design Verification

Simulation is the principle verification tool in HDL. Furthermore, most Co-design verification methods depend on Co-simulation of two or more types of components that are designed by different technologies. Each research community tries to extend design stages to include more abstract levels. Fortunately, we can observe many common properties in the research result of these different communities. It is quite interesting to compare them and to show that they could be prefigured and structured within a model driven architecture. In this paper, we focus on development with B approach and show how it may be applied on HDL.

4 B METHOD, MOCHA, EVENT B:

B method [1] is known in software engineering as a formal method to specify and to develop finely the specification towards an executable program basing on set theory and first order logic notation. B draws together advances in formal methods that span the last forty years (pre and post notations, guarded commands, stepwise refinement, and the refinement of both calculus and data). During the software development in B method, many versions of the same component may be found. The first and the most abstract one is the Abstract Machine where client needs are declared. Then the following versions should be more concrete and precise more and more how we obtain the needed specifications. These versions are called Refinements except the last one where there is no more possible refinement. This deterministic version is called Implementation. B generates the necessary proof obligations to verify the coherence of each component and correctness of the development. Furthermore, B tools help to execute these proofs.

Like B, Mocha [9] is an interactive verification environment for the modular and hierarchical verification of heterogeneous systems. Mocha

supports the heterogeneous modeling framework of reactive components and based on Alternating Temporal Logic (ATL), for specifying collaborations and interactions between the components of a system. Event B is an evolution of B Method. Key features of B Event are the extensions to events for modeling concurrency. The primary concept in doing formal developments in Event-B is that of a model. A model contains the complete mathematical development of a Discrete Transition System. It is made of several components of two kinds: machines and contexts. Machines contain the variables, invariants, theorems, and events of a model, whereas contexts contain carrier sets, constants, axioms, and theorems of a mode. The Rodin platform is an open source Eclipse-based IDE for Event B is further extendable with plugins.

5 BHDL: B ↔ VHDL

The principle of BHDL is to make use of the common properties between B, ADL and HDL in order to use a common formal iterance language. This will facilitate the verification of design correctness since the early steps of co-design. Fortunately, B method has its own mathematical notation that can be used during all development steps. The correctness of a system described by B language may be “proven” by many tools as AtelierB, BToolkit , B-For-Free and RODIN [3].

Declaration of ADL main components of system is graphically built, Then, two different notations are generated: VHDL and B.

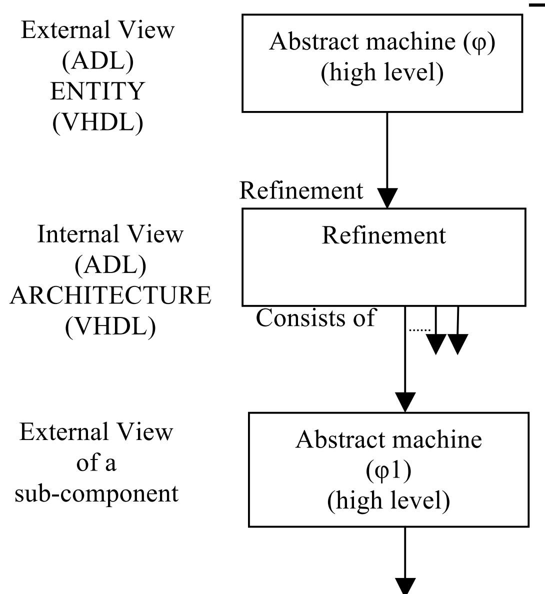


Figure 2: Common Aspects between ADL, HDL and B method.

The produced B code contains the main features of VHDL one. After that, design may be separated in relation to the technologic choices.

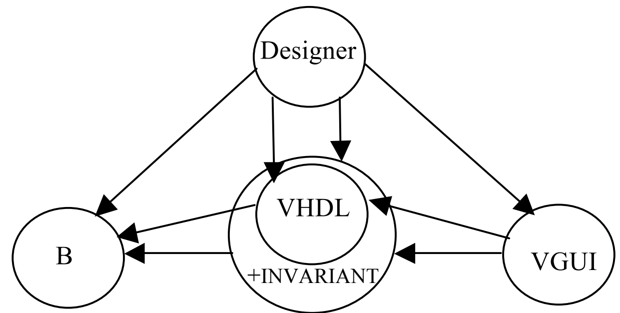
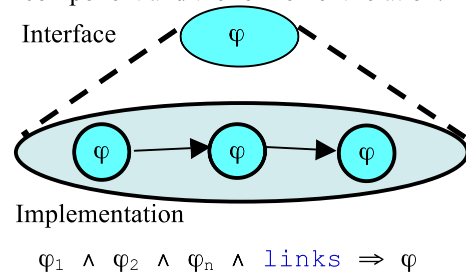


Figure 3: Principle of BHDL.

Each Architecture in VHDL is attached to one Entity and it may contain recursively one or more Entitys. This structure looks similar to extern-view and intern-view in ADL, procedure call and procedure implementation in imperative language .etc. Also in B method two basic components excite: the Abstract machine and the Refinement. The first one is usually used to precise the specifications of the component; the interface variables, the internal variables, the invariant relation between them and the pre and post conditions of the necessary operations. The second component may refine an abstract machine; that means it precise partly how the operations may be implemented. The Refinement component may be, in his turn, refined recursively by more deterministic Refinements. The last refinement step, when the behaviour becomes completely deterministic, is called the implementation. B tools may prove the consistency of each component and the refinement relation.



$\varphi_1 \wedge \varphi_2 \wedge \varphi_n \wedge \texttt{links} \Rightarrow \varphi$

Figure 4: Structural refinement and proof obligation

In our project each Entity is translated by an Abstract machine and each Architecture by a refinement. The ports are declared as Variables and the port typing as Invariant. Furthermore we enhanced the VHDL notation with logical properties. These properties are injected in B Invariant. The connection between subcomponents of the

Refinement should guarantee the Invariant specified in the abstract machine (see Fig. 4).

5.1 Hierarchy

In VHDL, the transition from an Entity into a corresponding Architecture is usually performed in one step. In BHDL, this may be finely performed by many steps or levels. We may consider the refinement of a component in BHDL as a replacement by other components. Also we may refine a component by another one which has the same structure and links but with more strict logic property. In all cases the refinement is performed towards lower levels where the behaviour of the system becomes more deterministic.

The principal relation between the interface (external view) and its refinement (or between two levels of refinement) is:

$$\text{Connection}(\varphi_{1,} \varphi_{2,}, \ldots \varphi_n) \Rightarrow \varphi$$

which means that the logical connection between the properties of the sub-components should satisfied the properties indicated in the abstract machine that represents the Entity.

5.2 Compositionality and Invariant

Let us consider the following simple example for illustrating captures of multiple mathematical views and reliability.

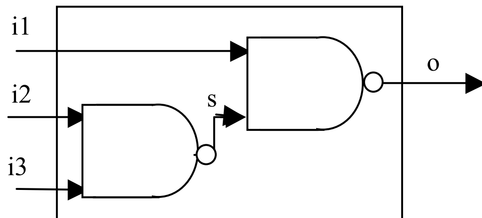


Figure 5: Structure of Comp1 component.

Fig. 5 shows a system that contains two Nand components. The modified version of VGUI allow to draw a similar connected boxes and to precise the logic properties and the internal structure of each box. Then VHDL+ and B code is generated.

VGUI generated the following VHDL+ code for this example:

```
STRUCTURE comp1 OF comp
SIGNAL s
BEGIN
  gate1 : nand  PORT MAP (i1,s,o)
  gate2 : nand  PORT MAP (i2,i3,s)
END
ENTITY nand
  PORT x, y : IN std_logic
       z : OUT std_logic
 -- z = nand (x,y) B specification
END
```

5.3 Specification Languages

As B is used in this example as formal specification language, PSL is an "add-on" language for Hardware description languages that has recently been standardized by the IEEE in 2005. PSL standard is based upon IBM's "Sugar" language, which was developed and validated at IBM Labs for many years before IBM donated the language to Accellera for standardization. PSL works alongside a design written in VHDL, Verilog or SystemVerilog. But in future it may be extended to work with other languages. Properties written in PSL may be embedded within the HDL code as comments or may be placed in a separated file alongside the HDL code. PSL includes multiple abstraction layers for assertion types ranging from low-level Boolean and Temporal to higher-level Modeling and Verification. Formally, PSL is structured into four layers: the Boolean, Temporal, Verification and Modeling layers. At its lowest-level, PSL uses references to signals, variables and values that exist in the design's conventional HDL description. Sugar used CTL (Computation Tree Logic) formalism to express properties for model checking. But the finally the underling semantic foundation was migrated from CTL to LTL (Linear-Time Temporal Logic) because the latter is considered more accessible to a wider audience and it is more suitable for simulation. The temporal operators of the foundation language provide syntactic sugaring on the top of LTL operators. These temporal operators include:

Always: it holds if its operator holds in every signal cycle.

Never: it holds if its operand fails to hold in every signal cycle.

Next: it holds if its operand holds in the cycle that in the immediately follows.

Until: it holds if the property at its left-hand holds in every cycle from the current cycle up until the next cycle in which the property at its right-hand holds.

Before: it holds if the left-hand operand holds at least once between the current cycle and the next time the right-hand operand holds.

5.4 Fault Tolerance in BHDL

The usual development in B method goes from the abstract requirement to the concrete execution. During the development, the behaviour becomes more and more deterministic. In spite of that, BHDL can takes in account the possibility to describe a fault scenario. Here we describe the ideal system with the behaviour of the ideal variables in the abstract machine, then, by Refinement, we inject the possible fault. This fault is declared using false variables. Then, we propose the correction step for the false variables. At the end, we prove that the corrected

values of the false variables respect the INVARIANT of the initial ones. The additional variables and the correction operations are the cost of trust behavior of the system.

5.5 Dependency Relation

BHDL project can make use of B tools to verify the dependence between an output and an input. In Refinement components, each connection produces an independency relation between two variables. Two types of connections may be noticed; the connection between the sub-components and the intern wires and the connection between sub-components and outer ports.

The direction of the dependency is related to the signal direction. As we see, this relation recursively depends on the lower levels. As Refinement (architecture) can see only the abstract machines (ENTITYs) of its sub-components. So that, as the Refinement can not see the Refinements of its own sub-components, it cannot see their dependency relation (see Fig. 6). One solution is to modify the Invariant of each Abstract machine where dependency relation is declared. To facilitate the modification we write a part the invariant of the abstract machine in an independent file that may be easily modified by the refinement.

We defined a transitive relation "Depend" on the ensemble PORTS with one direction. This relation should be defined on variables attached to the instances of the interne components not to the generic form of them so we add new variables for each instance to define the dependency relation. For example, we shall write the dependency relation for the following component.

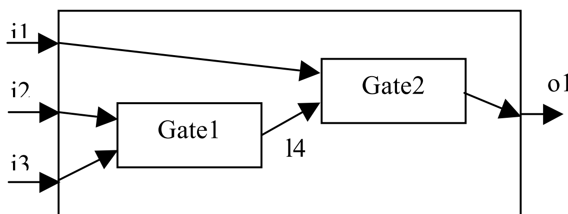


Figure 6: Dependency Relation.

All these modification of the INVARIANT are applied at refinement level where we can see the subcomponents. But we need this information at the abstract machine level because we need to know the dependency relation in a higher level where this component (or abstract machine) is included, in its turn, as subcomponent. The abstract machine of the right part of Fig. 7 is used as a sub-component in the refinement of the left part.

This dependency relation has been use to check fan-out property. In digital circuits, fan-out defines the maximum number of digital inputs that the output of a single logic gate can feed. The value of the fan-out is a big impact on test and debugging.

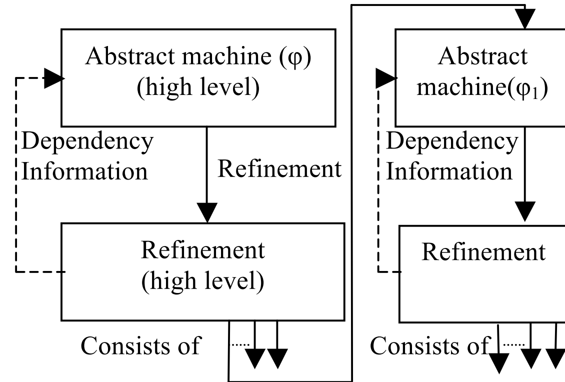


Figure 7: Dependency Information Transfer.

6 REALISATION OF BHDL PROJECT:

The project is totally implemented by three distinct components of BHDL:

6.1 A Graphical Interface for System Entry (VGUI)

As we mentioned above, we make use of VGUI (VHDL Graphical User Interface) to built the system entry of Hardware Diagrams. It is an open source tool that may be considered as a simple component description tool. VGUI may be used to create generic interconnected boxes. Each box may be decomposed hierarchically into sub-boxes and so on. The boxes and the connections of VGUI are typed. In cooperation with VGUI developer, we added the possibility to attach logic property to each box and hide data. Eventually, VGUI generates VHDL code annotated with B expressions. This step is optional; designer may use a textual editor to directly write the annotated code to be analyzed by the following step.

6.2 B Model Generator

Here a B model that corresponds to the annotated VHDL model is crated. The A complier is built to generate B code. From the external view of VHDL or from an entity in VHDL model, it generates the suitable B Abstract Machine that contains the necessary properties of the Entity and traces the structure of VHDL model.

In a similar way, the internal view in VGUI is translated into Architecture in VHDL then into a refinement in B. Because that design in VHDL

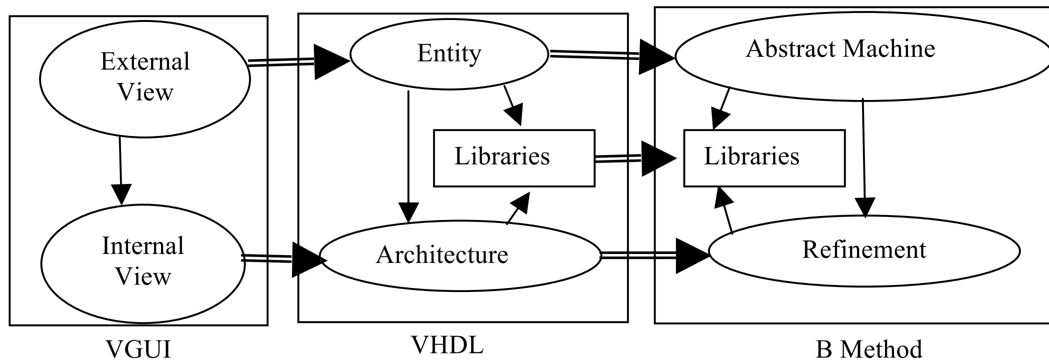


Figure 8: main transformations of BHDL.

usually depends of some predefined standard libraries, we created some B components that correspond to some VHDL libraries (such as the Standard logic 1164).

The compiler is the most important practical part of BHDL project. It is built on ANTLR compiler generator. ANTLR (Another Tool for Language Recognition) is a powerful tool that accepts grammatical language descriptions and generates programs (compliers or translators) that can recognize texts in the described languages, analyzes these texts, constructs trees corresponding to their structure and generates events related to the syntax. These events, written in C++ or in Java, may be used to translate the text into other languages. It can generate AST (Abstract Syntactic Trees) which can stock a lot of information about the analyzed text, provides tree rewriting rules for easily translating these ASTs. The correction of such a translator depends only on the correction of every elementary rewriting rule (declarative semantics). As VGUI, ANTLR is open source software written in Java. The translation from VHDL+ to B in is performed over many steps:

- BHDL Lexer/Parser : which analyses the input VHDL+, verifies the syntax and the semantic of VHDL code, then it generates a pure VHDL tree (AST) with independent branches that contain the B annotations
- TreeWalker: this tree parser parses the previous AST in order to capture the necessary information to construct a new AST that corresponds to B model.
- B-Generator: It traverses the AST produced by the TreeWalker in order to generate B code.

Even if a corresponding B model is automatically created, the design correctness is not automatically proven. The generated B code should be proven to be correct. B tools (AtelierB, B4Free, B-Toolkit) render the task easy. It generates the necessary prove obligations (POs), automatically produces an important quantity of the proofs, cooperates with the programmer to prove the rest of the POs. Here, if the model is not completely proven, some defects may be detected and the original VHDL design should be modified.

7 AFCIM AND PCSI PROJECTS

BHDL project is developed in the LIFL (Lille's Computer Science Laboratory). This research first conducted into the AFCIM project (LIFL, INRETS, HEUDIASYC Lab).

The French project AFCIM (Formal Architectures for Conception and Maintenance of Embedded Systems) coordinated by Philippe Devienne (LIFL) is a collaborative research between four French universities and institutes (LIFC, LIFL, Heudiasyc, INRETS).

The global architecture of the AFCIM project is shown in Fig. 9:

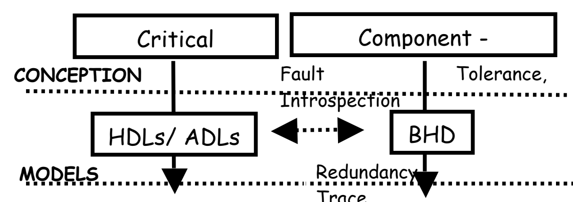


Figure 9: AFCIM Project

From a general Model Driven Architecture (i.e the common part of specific description languages like ADL, HDL...), we add formal annotations and specifications according to the requirements or the fault scenarios that we want to handle. All the tools used in our platform are freely used and distributed (Rodin, Eclipse, Antlr, …).

Eventually the main concepts of BHDL and AFCIM is being augmented and implemented with support of PCSI project (Zero Defect Systems) between Lille University, Aleppo University and

Annaba University. The main new features of the project are the following Fig.10:

7.1 Including PSL

Instead of special comments used in the first version of BHDL to represent the logical behavior of VHDL components, we use here a formal language, PSL, that is standardized in 2005. PSL (Property Specification Language) [12] is a language for the formal specification of hardware. It is used to describe properties that are required to hold in the design under verification. It contains Boolean, Temporal, Verification and modelling layers. The flavour of PSL could be added to many HDL (Hardware Description Language) such as VHDL, Verilog, SyetemVerilog. This enlarges the usability of our tool since PSL is expressive and standard.

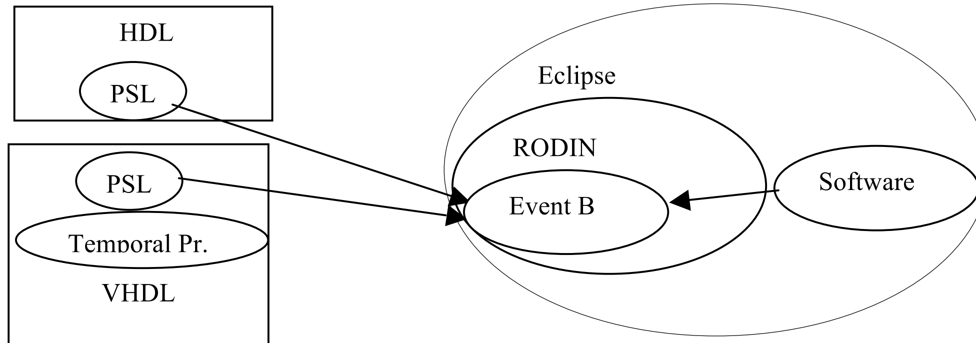


Figure 10: basic augmentation in the PCSI project (Zero Defect Systems) vs BHDL.

7.2 Extending Scope of VHDL Treated in BHDL

While the first version of BHDL mainly manipulates the design structure decorated with logical properties, here we enlarge the model to accept important concepts of VHDL such as signals where the concept of Time appears.

Beside ENTITY and ARCHITECTURE VHDL contains other design units such as CONFIGURATION. These units could be taken in the future.

7.3 Creating the Target Model Using Event-B Instead of Classical B

The purpose of Event-B is to model full systems (including hardware, software and environment of operation). Classical B is not suitable to represent temporal properties which are important in hardware design. Furthermore, Event-B facilitates the representation of many subsystems in a global one.

After the creation of an HDL model, it will be traced in B. in order to facilitate the proof of the consistency and the formal refinement of the model; we integrated our work in Eclipse environment. Eclipse is generic platform to develop multi-language software comprising an integrated development environment (IDE) and an extensible plug-in system. The Rodin Platform is an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof. The platform is open source, contributes to the Eclipse framework and is further extendable with plugins. Such integration renders the integration between hardware community and software community easy since they work on the same environment. All the tools used in our platform are freely used and distributed (Rodin, Eclipse, Antlr, …).

7.4 Automated Addition of Robustness

We focus on the problem in evolving a fault-intolerant program to a fault-tolerant one. The question is "Is It possible to add a default scenario to an existing model or program and generate automatically the tolerant model or program?" This problem occurs during program evolution new requirement (fault-tolerance property, timing constraints, and safety property) change. We argue here that refinement can handle this evolution. In others words a fault-tolerant program is a refined form of its intolerant one. We have shown how to apply this formalism to characterize fault-tolerance mechanisms and to then reason about logical and mathematical properties. For instance, the hamming code is a kind of data refinement. By adding data redundancy (extra parity bits), error-detection and even error-correction are possible. This can generalize to handle Byzantine properties.

Fault tolerance is often based on replication and redundancy. This is involved by the use of hybrid systems with different sources of energy (electric, mechanic). This duplication can be also seen as component refinement or algorithmic refinement. For instance, nowadays, because of the integration of circuits, stuck-at–fault is a more and more frequent fault model. According that the probability that a circuit contains at least k stuck-a-fault is too high, we can generate an equivalent circuit, except that it is k-stuck-at-fault tolerant. This transformation can be seen a refinement, that a logico-mathematical completion w.r.t. a default model.

ACKNOWLEDGEMENT
The CAD tool, VGUI, is adapted to or project in cooperation with Mr. Carl Hein.
An ANTLR parser template to generate AST trees is built in cooperation with Mr. J. L. Boolanger.

8 REFERENCES

[1] J.-R. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge University Press, UK, 1996.
[2] Y. Herve, VHDL-AMS – Applications et enjeux industriels, Duand, France, 2002.
[3] The website of Event B and RODIN [Online]. Available http://www.event-b.org/platform.html
[4] Flaviu Cristian, Understanding Fault-Tolerant Distributed Systems, ACM, February 1991, 34(2): 56-78
[5] Terence Parr, The definitive ANTLR Reference, May, 2007
[6] Ammar Aljer, Co-design and refinement in B, Ph.D. Thesis, Lille Computer Science Laboratory, Lille, France, Dec. 2004.
[7] D. Garlan, Formal Modeling and Analysis of Software Architecture: Components, Connectors and Event, Springer-Verlag, Sep 2003.
[8] I. Sommerville, *Software Engineering*, Pearson, 2007
[9] Rajeev Alur, et al. Mocha: Modularity in model checking. In Proceedings of the Tenth International Conference on Computer-aided Verification, Lecture Notes in Computer Science 1427, Springer-Verlag (1998).
[10] Kenneth E. Kendall et Julie E. Kendall, Systems Analysis and Design, 8/E, Prentice Hall (2010).
[11] Anish Orora, Gray-Box Component-Based Fault-Tolerance, Logical Aspects of Fault Tolerance (LAFT), a LICS 2009 Workshop.
[12] DOULOS, PSL Golden Reference guide, Book, 2005.