Formal and Fault Tolerant Design

Ammar Aljer, Philippe Devienne

FORMAL AND FAULT TOLERANT DESIGN

Ammar Aljer[1], Philippe Devienne [2]

[1]*Faculty of Electrical and Electronic Engineering, University of Aleppo, Aleppo, Syria*
[2] *Lille's Computer Science Laboratory, University of Lille, Lille, France*

Abstract: Software quality and reliability were verified for a long time at the post-implementation level (test, fault scenario …). The design of embedded systems and digital circuits is more and more complex because of integration density, heterogeneity. Now almost ¾ of the digital circuits contain at least one processor, that is, can execute software code. In other words, co-design is the most usual case and traditional verification by simulation is no more practical. Moreover, the increase in integration density comes with a decrease in the reliability of the components. So fault detection, diagnostics techniques, introspection are essential for defect tolerance, fault tolerance and self repair of safety-critical systems.
The use of a formal specification language is considered as the foundation of a real validation. What we would like to emphasize is that refinement (from an abstract model to the point where the system will be implemented) could be and should be formal too in order to ensure the traceability of requirements, to manage such development projects and so to design fault-tolerant systems correct by proven construction. Such a thorough approach can be achieved by the automation or semi-automation of the refinement process.
We have studied how to ensure the traceability of these requirements in a component-based approach. Reliability, fault tolerance can be seen here as particular refinement steps. For instance, a given formal specification of a system/component may be refined by adding redundancy (data, computation, component) and be verified to be fault-tolerant w.r.t. some given fault scenarios. A self-repair component can be defined as the refinement of its original form enhanced with error detection.
We describe in this paper the PCSI project (Zero Defect Systems) based on B Method, VHDL and PSL. The three modeling approaches can collaborate together and guarantee the codesign of embedded systems for which the requirements and the fault-tolerant aspects are taken into account for the beginning and formally verified all along the implementation process.

I INTRODUCTION

The final decades of the 19th century and the first decades of the 20th century witnessed many efforts to formulate mathematics. Some fruits of these essays are Set theory, Propositional Logic, First Order Logic, etc. Introduced by Alonzo Church in the 1930s, λ-calculus is a primitive method to formalize algorithms where many concepts similar to those of programming languages are well defined such as: Recursion and fixed points, Logic and predicates, Free and bound variables, Substitutions. In the beginning of 1950s Von Neumann described a computer architecture in which the data and the program are both stored in the computer's memory in the same address space. This architecture is to this day the basis of modern computer design. In the beginning programmers wrote their programs as strings of zeros and ones. A work would often be an extremely frustrating activity. Rapidly this task is facilitated depending on Assembly language and OpCode tables. Developed in the mid-1950s, FORTRAN was intended for use in scientific and numerical computing applications. It 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 programing then OOP) concentrated on the abstraction of the executive machine code. Nowadays writing the implementation is partially automated and designer may give more attention on system structure. Actually with CASE (computer Aided Software Engineering) tools and with techniques such as MDA (Model Driven Architecture), programmer can graphically specify the components of the 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. Verification becomes more and more difficult because its cost increases exponentially with complexity. Reusing is another aspect of complex systems. In most cases 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. He ought to adapt them to the novel environment.
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 engi-

This work was partially supported by the PCSI program *Zero Defect Systems* (http://www.auf.org/anglais)

neering developed many principles and techniques to formulate client requirements. Comparing to the development of programming language, these efforts rest primitive and a formal gap between what a program do and what a client wants is always exists.
B method (1996) filled partially the gap. It defines what a formal refinement of software is. So it guarantees the complete correctness of software regarding to its formal specification. In our approach this method is generalized to be used in software, hardware and in embedded systems. Proving the correctness of one component is usually expensive comparing to the traditional methods of verification but this is rapidly compensated when the component is reused and when complexity augments; proving the correctness of a system that consists of proven components needs only to prove the correctness of the connections between the components. This approach also facilitates the verification parallelism since each component could be independently proven.
Components in real word (especially hardware ones) do not correspond 100% to their formal specification. This is a cause for many failures in the system even if it was proven to be correct or if its behavior is verified during the simulation. Another important feature of our approach is the possibility to prove the correctness of model even with real failure scenario if it is combined with a suitable correcting treatment.

II DOMAIN SPECIFIC LANGUAGES

On the opposite of programming languages who are designed for experimented programmers, a Domain Specific Language (DSL), comes from a domain and is used by users of this domain. Thus, a successful DSL is of course a used language and first intuitively usable by users of the chosen domain.

One of the first Domain Specific Language (DSL) was introduced for children. Its name was Logo and was designed by Seymour Papert at MIT in the sixties. He was been nominated by Marvin Minsky as "the greatest living educator in Mathematics". In Mindstorms [5], Seymour Papert explained that some children have difficulties in mathematics logic and this new language was specifically create to improve the way that children solve mathematical problems. Excel and MatLab are two well-known examples of DSL in mathematics too. Excel was even described as a *killer application* because it is so easy and funny to use by anyone.

This is quite opposite to view of universality in general-purpose programming language, such as C or Java, or a general-purpose modeling language such as UML.

Recently, the DSL approach has really been successful in two domains, web applications and cell phones. There are a lot of View/Edit WebDSLs from which we can generate Java or PHP code, web pages and Seam session beans. For instance, SPIP is a publishing system for the Internet in which great importance is attached to collaborative working, to multilingual environments, and to simplicity of use for web authors
Developing a new DSL needs definitively a good understanding of the application domain, then the next usual consist of finding programming patterns, designing a core language, building syntactic abstractions on top of the core language. But this type of design is a real complex activity and must be based on good tools, especially for verification. A lot of researches have to be lead to propose such an appropriate environment with good tools and libraries.

From the other hand, another challenge is appeared with embedded systems where more or more communities (usually hardware and software) with totally different methodologies, terminologies and measurements should design one common component! The DSLs have to be combined and collaborate in the same final objet.

A. VHDL

Due to the difference between hardware product and software product, Production of hardware or software component passes through two 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. VHDL that is the dominant language in hardware design was the first to take system level in account.
Even if VHDL 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 behavior of the circuit instead of its structure.

B. B METHOD, MOCHA, B-Event

B method [1,2] 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, the refinement calculus and data refinement). 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 [3] is a 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 (section 2) 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. The overall objective of this open platform is to propose a toll for the cost effective rigorous development of dependable complex systems and services. It focus on tacking complexity (1) caused by the environment in which the software I to operate (2) which comes from poorly conceived architectural structure. Mastering complexity in the shortest time-to-market requires design techniques that support clear thinking and rigorous validation and verification. Coping with complexity also requires architectures that are tolerant of faults and unpredictable changes in environment. This is addressed by fault tolerance design techniques.

III FORMAL HARDWARE DESIGN IN BHDL PROJECT

In this section, we are going to focus of the collaboration of two DSL languages, one for mathematics and logic (Event B) and the other to design VLSI (VHDL). Both have been widely validated by complex large industrial applications. Here we try to combine the advanced notion of formal refinement in B with the formal conception of HDL. The core of the work, from which the name of the project BHDL comes, is to create the correspondence between a VHDL design and a B one. In VHDL, the transition from an Entity into a corresponding Architecture is usually performed in one step. In BHDL, this may be performed finely 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 behavior of the system becomes more deterministic.

The principal relation between the interface (external view) and its refinement (or between two levels of refinement) is $Connection(\varphi 1, \varphi 2, \ldots \varphi n) \Rightarrow \varphi$ 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. The property (φ) in the interface is not original in VHDL, it is inserted in special comments in VHDL code so BHDL code does not affect the design portability.
The principle of B refinement permits not only to prove the consistency of Architecture but also to prove the correctness of the design w.r.t. the abstract specification in the external view (VHDL Entity). Furthermore it allows hardware community to built their pattern throw many smooth phases instead of one rough phase from all the components should be specified. The main components of BHDL project are:

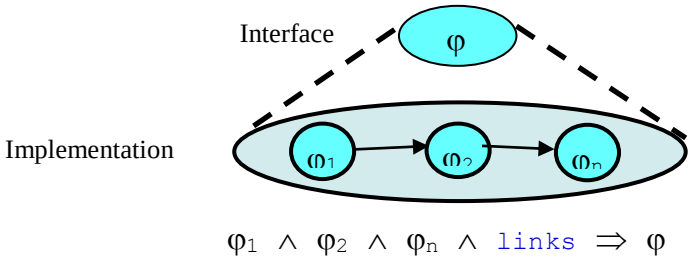

Figure 1: Structural refinement and proof obligation.

A. *A graphical interface for System Entry (VGUI)*

It is a Graphical User Interface for 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.

B. *B Model Generator*

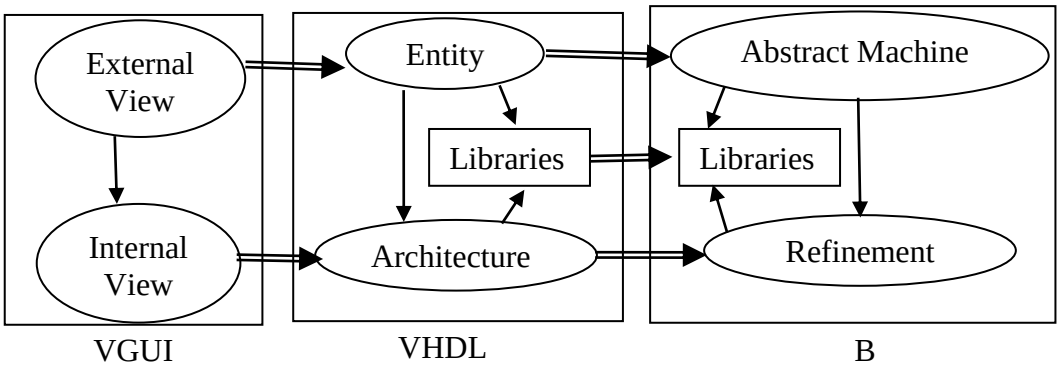

Figure 2: main transformations of BHDL.

Here a B model that corresponds to the annotated VHDL model is created. The ANTLR compiler is used 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 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, Rodin, B4Free, B-Toolkit) render the task easier. It generates the necessary proof obligations (POs), automatically produces an important quantity of these proofs, cooperates with the programmer to prove the remaining POs. Here, if the model is not completely proven, some defects may be detected and the original VHDL design should be modified.

IV PCSI PROJECT

BHDL project is developed in the LIFL (*Lille's Computer Science Laboratory*). This research first conducted into the AFCIM project (LIFL, INRETS, HEUDIASYC Lab). Eventualy the main concepts of BHDL have been extended 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:

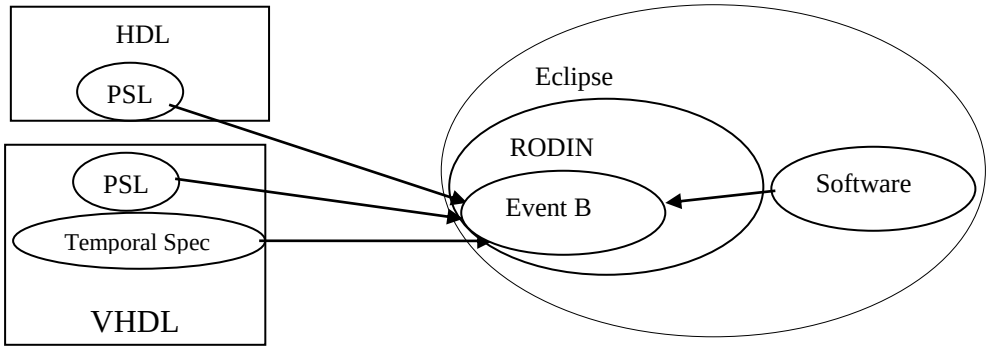

Figure 3: basic augmentation in the PCSI project (Zero Defect Systems) vs. BHDL.

A. 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, standardized in 2005. PSL (Property Specification Language) is a language for the formal specification of hardware. This language is essentially based on Linear Temporal Logic "LTL". 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, SystemVerilog. This enlarges the usability of our tool since PSL is expressive and standard.
In this project, we generate a software model which is B representation of a model described by Property Specification Language "PSL" using Event B Systems. Generated model can be proven by using B method techniques; this means a proof of the original PSL model.

B. 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.

C. 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 a 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, …).

D. Completion wrt robustness

Robustness or Fault-Tolerance is by definition "The ability of a system to respond gracefully to an unexpected hardware or software failure". There are many levels of fault tolerance, the lowest being the ability to continue operation in the event of a power failure. Many fault-tolerant computer systems *mirror* all operations -- that is, every operation is performed on two or more duplicate systems, so if one fails the other can take over. Multitolerance refers to the ability of a system to tolerate multiple classes of faults. [19] illustrates a compositional and stepwise method for designing programs and handling Byzantine failures [18]. It proposes also a component based method, starts with a intolerant system and adds a set of components, one for each desired type of tolerant. The complexity of multitolerant design is reduced to that of designing the components and of correctly adding them to the intolerant one.

Indeed, 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. We focus here 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 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 "minimal" + data refinement. By adding data redundancy (extra parity bits), error-detection and even error-correction are possible. This can generalize to handle Byzantine properties. For instance, *masking tolerance* considers that in the presence of faults each step in the system computation satisfies *Validity* and *Agreement* properties. Weakly, *Stabilizing tolerance* considers that each step in the system computation will satisfy in a near future (reachable state) Validity and Agreement. What we show here is that fault-tolerant design is compositional (component-based method), adaptive (stepwise and hierarchical approach) and formal (these completions are refinements which have to be formally proven). The fault-tolerant design appears as a logico-mathematical completion of an intolerant model in order to tolerant multiple classes of faults.

V CONCLUSION

The novel aspects of this proposal are the pursuit of a process-based approach, the combination of Formal Methods with Fault Tolerant techniques, the development of FM support for component reuse and composition and the extension of an open and extensible tools platform for formal development. It is clear that the open tools platforms will have a more and more important impact on future research, but they have to be adapted to users and their languages as VHDL. We believe that proposing intelligent interfaces between DSL approaches (as VHDL+PSL) and FM tools (as RODIN) is the shortest way to make these techniques more popular and will encourage greater industrial uptake.

REFERENCES

[1] Abrial J.R. 1996, The B-Book: assigning programs to meanings. UK: Cambridge University Press.

[2] Abrial J.R, 2010, Modeling in Event-B, System and Software Engineering. UK: Cambridge University Press.

[3] Rajeev Alur, et al. 1998. Mocha: Modularity in model checking. In Proceedings of the Tenth International

[4] Conference on Computer-aided Verification, Lecture Notes in Computer Science 1427, Springer-Verlag.

[5] Seymour Pappert, 1980, Mindstorms: Children, Computers, and Powerful Idea

[6] Ammar Aljer, Co-design and refinement in B, Ph.D. Thesis, Lille University, 2004.

[7] Philippe Devienne , Rabih Oueidat, Formal Tolerant Software/Harware Architecture, In Specification and Verification of Component-Based Systems, Workshop at SIGSOFT 2008/FSE 16 (SAVCBS 08), Atlanta, Nov 2008

[8] Philippe Devienne, Ammar Aljer, Extended Model Driven Architecture to B method, Ubiquitous Computing and Communication Journal, 2011

[9] Y. Herve, VHDL-AMS – Applications et enjeux industriels (in french), Durand, France, 2002.

[10] RODIN : http://www.event-b.org/platform.html

[11] Flaviu Cristian, Understanding Fault-Tolerant Distributed Systems, ACM, Feb 1991, 34(2): 56-78

[12] Terence Parr, The definitive ANTLR Reference, 384 pages, May 2007

[13] DOULOS, PSL Golden Reference guide, Book, 2005.

[14] Anish Orora, Gray-Box Component-Based Fault-Tolerance, Logical Aspects of Fault Tolerance (LAFT), a LICS 2009 Workshop.

[15] Kenneth E. Kendall et Julie E. Kendall, 2010, Systems Analysis and Design, 8/E, Prentice Hall.

[16] Ian Sommerville, 2008, Software Engineering: International Version , 8/E, Addison-Wesley

[17] The Rascal Domain Specific Language, INRIA Report, 2009

[18] L. Lamport, R. Shostak and M.Pease. The Byzantine generals problem. ACM Transactions on Programming Languages and Systems, 1982.

[19] S.S. Kulkarni, A.Arora, Composition Design of Multitolerant Repetitive Byzantine Agreement, 17th Conference on Foundations of Software Technology and Theoretical Computer Science, 1997