Luboš Brim  Boudewijn Havorkort
Martin Leucker  Jaco van de Pol (Eds.)

Formal Methods:
Applications
and Technology

11th International Workshop, FMICS 2006
and 5th International Workshop, PDMC 2006
Bonn, Germany, August 26-27, and August 31, 2006
Revised Selected Papers
Volume Editors

Luboš Brim
Masaryk University
Botanicka 68a, 602 00 Brno, Czech Republic
E-mail: brim@fi.muni.cz

Boudewijn Haerkort
University of Twente
P.O. Box 217, 7500AE Enschede, The Netherlands
E-mail: brh@cs.utwente.nl

Martin Leucker
Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
E-mail: leucker@in.tum.de

Jaco van de Pol
Centrum voor Wiskunde en Informatica, SEN 2
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
E-mail: Jaco.van.de.Pol@cwi.nl

Library of Congress Control Number: 2007921124


LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com
© Springer-Verlag Berlin Heidelberg 2007
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12021901 06/3142 5 4 3 2 1 0
Preface

These are the joint final proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2006) and the fifth International Workshop on Parallel and Distributed Methods in Verification (PDMC 2006). Both workshops were organized as satellite events of CONCUR 2006, the 17th International Conference on Concurrency Theory that was organized in Bonn, August 2006.

The FMICS workshop continued successfully the aim of the FMICS working group – to promote the use of formal methods for industrial applications, by supporting research in this area and its application in industry. The emphasis in these workshops is on the exchange of ideas between researchers and practitioners, in both industry and academia.

This year the Program Committee received a record number of submissions. The 16 accepted regular contributions and 2 accepted tool papers, selected out of a total of 47 submissions, cover formal methodologies for handling large state spaces, model-based testing, formal description and analysis techniques as well as a range of applications and case studies.

The workshop program included two invited talks, by Anna Slobodova from Intel on “Challenges for Formal Verification in an Industrial Setting” and by Edward A. Lee from the University of California at Berkeley on “Making Concurrency Mainstream.” The former full paper can be found in this volume.

Following the tradition of previous workshops, the European Association of Software Science and Technology (EASST) supported a best paper award. This award was granted to Michael Weber and Moritz Hammer for their excellent paper “‘To Store or Not To Store’ Reloaded: Reclaiming Memory on Demand.”

The primary goal of the PDMC workshop series is to present and discuss recent developments in the young area of parallel and distributed methods in verification. Several verification techniques, ranging over model checking, equivalence checking, theorem proving, constraint solving and dependability analysis are addressed by the PDMC community. Verification problems are usually very demanding tasks, especially because the systems that we build and want to verify become increasingly complex.

On the other hand, parallel and distributed computing machinery is widely available. Algorithms and tools must be developed to use this hardware optimally for our verification tasks. Traditionally, we studied algorithms for homogeneous situations, such as parallel shared-memory computers and distributed clusters of PCs. Currently, the emphasis is shifting towards heterogeneous GRIDs. But even modern desktop PCs are quite heterogeneous, consisting of multiple core processors, various memory devices and cache levels, all with their own performance characteristics.
This year’s PDMC had nine submissions; six papers were selected for presentation, and four papers were accepted for publication in this volume. In addition, Luboš Brim from Masaryk University, Brno, gave an invited lecture on “Distributed Verification: Exploring the Power of Raw Computing Power.” The full paper can also be found in this volume.

We would like to thank all authors for their submissions. We would also like to thank the members of both Program Committees, and the additional referees, for their timely reviewing and lively participation in the subsequent discussion—the quality of the contributions in this volume are also due to their efforts and expertise.

The organizers wish to thank CONCUR for hosting the FMICS and PDMC 2006 workshops and taking care of many administrative aspects, and ERCIM for its financial support of FMICS. Additionally, the organizers would like to thank the EASST (European Association of Software Science and Technology), the Faculty of Informatics, Masaryk University Brno and the Technical University Munich, the CWI (Center of Mathematics and Computer Science, Amsterdam) and the University of Twente for supporting these events.

December 2006

Luboš Brim
Boudewijn R. Haverkort
Martin Leucker
Jaco van de Pol
Organization

FMICS

Program Chairs
Luboš Brim  
Masaryk University Brno, Czech Republic
Martin Leucker  
Technical University of Munich, Germany

Program Committee
Rance Cleaveland  
University of Maryland, USA
Wan Fokkink  
Vrije Universiteit Amsterdam and CWI, The Netherlands
Stefania Gnesi  
ISTI-CNR, Italy
Susanne Graf  
VERIMAG, France
David Harel  
Weizmann Institute of Science, Israel
Klaus Havelund  
Kestrel Technology, USA
Thomas A. Henzinger  
EPFL, Switzerland
Leszek Holenderski  
Philips Research, The Netherlands
Stefan Kowalewski  
RWTH Aachen University, Germany
Marta Kwiatkowska  
University of Birmingham, UK
Salvatore La Torre  
Università degli Studi di Salerno, Italy
Tiziana Margaria  
University of Göttingen, Germany
Radu Mateescu  
INRIA Rhône-Alpes and ENS Lyon, France
Doron Peled  
University of Warwick, UK
Ernesto Pimentel  
University of Malaga, Spain
Andreas Podelski  
Max-Planck-Institut für Informatik, Germany
Don Sannella  
University of Edinburgh, UK
Joseph Sifakis  
VERIMAG, France

PDMC

Program Chairs
Boudewijn Haverkort  
University of Twente, The Netherlands
Jaco van de Pol  
CWI Amsterdam, The Netherlands

Program Committee
Gerd Behrmann  
Aalborg University, Denmark
Ivana Černá  
Masaryk University Brno, Czech Republic
Gianfranco Ciardo  
University of California at Riverside, USA
Joerg Denzinger  
University of Calgary, Canada
Hubert Garavel
Orna Grumberg
William Knottenbelt
Marta Kwiatkowska
Martin Leucker

INRIA Rhône-Alpes, France
Technion, Haifa, Israel
Imperial College, London, UK
University of Birmingham, UK
Technical University of Munich, Germany

Referees (FMICS and PDMC)

C. Artho   I. Černá   M. Kuntz   D. Parker
Y. Atir     F. Ciesinski  F. Lang   G. Parlato
R. Atkey    M. Faella   P. Lopez   G. Salaün
J. Barnat   A. Fantechi  K. MacKenzie  W. Serwe
M. ter Beek  M. Felici  P. Maier   F. Sorrentino
M. van der Bijl  A. J. Fernandez  S. Maoz   J. Tenzer
B. Bollig  M. Fruth   F. Mazzanti  A. Venet
L. Bozzelli   N. Geisweiller  R. Merom   A. Wijs
A. Bucchiarone  A. Goldberg  A. Murano   T. Willemse
D. Calvanese   A. Idani   G. Norman   V. Wolf
M. V. Cengarle  C. Joubert  M. Parente
# Table of Contents

## Invited Contributions

- Challenges for Formal Verification in Industrial Setting .......................... 1  
  *Anna Slobodová*
- Distributed Verification: Exploring the Power of Raw Computing Power .......................... 23  
  *Luboš Brim*

## FMICS

- An Easy-to-Use, Efficient Tool-Chain to Analyze the Availability of Telecommunication Equipment .......................................................... 35  
  *Kai Lampka, Markus Siegle, and Max Walter*
- “To Store or Not To Store” Reloaded: Reclaiming Memory on Demand .................................................. 51  
  *Moritz Hammer and Michael Weber*
- Discovering Symmetries ................................................................. 67  
  *Hassen Saïdi*
- On Combining Partial Order Reduction with Fairness Assumptions ............... 84  
  *Luboš Brim, Ivana Černá, Pavel Moravec, and Jiří Šimša*
- Test Coverage for Loose Timing Annotations ........................................ 100  
  *C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz*
- Model-Based Testing of a WAP Gateway: An Industrial Case-Study ............. 116  
  *Anders Hessel and Paul Pettersson*
- Heuristics for ioco-Based Test-Based Modelling ...................................... 132  
  *Tim A.C. Willemse*
- Verifying VHDL Designs with Multiple Clocks in SMV .............................. 148  
  *A. Smrčka, V. Řehák, T. Vojnar, D. Šafránek, P. Matoušek, and Z. Řehák*
- Verified Design of an Automated Parking Garage .................................... 165  
  *Aad Mathijssen and A. Johannes Pretorius*
- Evaluating Quality of Service for Service Level Agreements .................... 181  
  *Allan Clark and Stephen Gilmore*
Simulation-Based Performance Analysis of a Medical Image-Processing Architecture ................................................................. 195  
*P.J.L. Cuijpers and A.V. Fyukov*

Blasted Linux Code ............................................ 211  
*Jan Tobias Mühlberg and Gerald Lüttgen*

A Finite State Modeling of AFDX Frame Management Using Spin ..... 227  
*Indranil Saha and Suman Roy*

UML 2.0 State Machines: Complete Formal Semantics Via Core State Machines .................................................................................. 244  
*Harald Fecher and Jens Schönborn*

Automated Incremental Synthesis of Timed Automata .................. 261  
*Borzoo Bonakdarpour and Sandeep S. Kulkarni*

SAT-Based Verification of LTL Formulas ................................. 277  
*Wenhui Zhang*

jmle: A Tool for Executing JML Specifications Via Constraint Programming ......................................................... 293  
*Ben Krause and Tim Wahls*

Goanna—A Static Model Checker ................................................ 297  
*Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch*

**PDMC**

Parallel SAT Solving in Bounded Model Checking ...................... 301  
*Erika Ábrahám, Tobias Schubert, Bernd Becker, Martin Fränzle, and Christian Herde*

Parallel Algorithms for Finding SCCs in Implicitly Given Graphs ...... 316  
*Jiří Barnat and Pavel Moravec*

Can Saturation Be Parallelised? – On the Parallelisation of a Symbolic State-Space Generator ....................................................... 331  
*Jonathan Ezekiel, Gerald Lüttgen, and Radu Siminiceanu*

Distributed Colored Petri Net Model-Checking with CYCLADES ...... 347  
*Christophe Pajault and Jean-François Pradat-Peyre*

**Author Index** ........................................................................... 363
Challenges for Formal Verification in Industrial Setting

Anna Slobodová

Intel
anna.slobodova@intel.com

Abstract. Commercial competition is forcing computer companies to get better products to market more rapidly, and therefore the time for validation is shrinking relative to the complexity of microprocessor designs. Improving time-to-market performance cannot be solved by just growing the size of design and validation teams. Design process automation is increasing, and the adoption of more rigorous methods, including formal verification, is unavoidable because for achieving the quality demanded by the marketplace.

Intel is one of the strongest promoters of the use of formal methods across all phases of the design development. Intel’s design teams use high-level modeling of protocols and algorithms, formal verification of floating-point libraries, design exploration systems based on formal methods, full proofs and property verification of RTL specifications, and equivalence checking to verify that transistor-level schematics correspond to their RTL specifications. Even with the best effort to adopt the progress in formal methods quickly, there is a large gap between an idea published at a conference and a development of a tool that can be used on industrial-sized designs. These tools and methods need to scale well, be stable during a multi-year design effort, and be able to support efficient debugging. The use of formal methods on a live design must allow for ongoing changes in the specification and the design. The methodology must be flexible enough to permit new design features, such as scan and power-down logic, soft error detection, etc. In this paper, I will share my experience with the formal verification of the floating-point unit on an Itanium(R) microprocessor design and point out how it may influence future microprocessor-design projects.

1 Introduction

Floating-point (FP) arithmetic is, with respect to functional validation, one of the critical parts of modern microprocessor designs. Even though the algorithms for FP arithmetic are well known, optimization for high performance, reliability, testability and low power, may introduce bugs into a design. The huge input data space that needs to be explored to ensure correctness of floating-point designs is beyond the limits of traditional simulation techniques (hereafter referred to as simulation). Fortunately, formal methods are well suited for this area and they can enhance a verification effort substantially. Formal semantics of floating-point
operations can be expressed in a succinct way and the IEEE Floating-Point standard [IEEE] serves as a guide for many instruction-set architectures. In this paper, floating-point algorithms are not our concern. Instead, I focus on the correctness of their register-transfer level (RTL) implementations.

For almost a decade, papers reporting compliance proofs for circuit models with respect to the IEEE standard and particular instruction set architecture have been published. While the early research was focused on answering a principal question of the feasibility of formal proofs for computer arithmetic (e.g., [AS95, CB98, OZ+99]), recent work emanates from commercial industry. Formal tools and methods have reached the maturity necessary for their deployment in real design projects.

There are substantial differences between the methodologies used at different companies, depending on their target and available tools and resources. Intel and AMD were among first companies that applied formal methods, at first to verification of floating-point algorithms and then to RTL design. Methods reported from AMD design team in [Rus98, RF00, FK+02] are solely based on theorem proving using ACL2 system 1. Although lot of automation has been added to building ACL2 models from RTL descriptions, and an ACL2 library of Floating-Point Arithmetic has been created to avoid repetition of implementation independent proofs, the methodology still requires high-level expertise in theorem proving and a perfect understanding of the design.

A recent paper from IBM by Jacobi et al. [JW+05] presents a verification method based on symbolic simulation of a RTL model and its comparison to a high-level model written in VHDL. Although highly automated, the approach is not as rigorous as the one described by AMD, and lacks the scope of the methodology developed at Intel [AJ+00][AJ+00, KA00, KN02, KK03, Sch03]. It skips the verification of the more difficult part of the design - multiplier, by removing it from the cone of influence, hence proving merely correctness of the adder and rounder. In contrast, in our approach, no abstraction or design modification took place. We return to the comparison of their work to our results in Section 7.

At Intel, an important work in the area of the verification of floating-point algorithms, and in particular, floating-point libraries for Itanium(R) has been done by John Harrison (see [Har05] for an overview, and [Har00a, Har00b, Har03] for details). However, in hardware verification, while many papers have been published on verification of Pentium(R) design, the first report on the formal verification of floating-point arithmetic for the Itanium(R) microprocessor family was reported on Designing Correct Circuits (DCC) Workshop 2, in March, 2004, in Barcelona [SN04]. The main result was the first successful formal verification of floating-point fused multiply-add instruction which is in repertoire of the IA-64 Instruction Set Architecture (ISA). Proofs have been constructed for a live project aimed at a next generation of Itanium(R) microprocessor. We continued our work presented at DCC by extending the scope and verifying correctness

1 http://www.cs.utexas.edu/moore/acl2/
2 http://www.math.chalmers.se/~ms/DCC04/
of the rest of floating-point instructions (about 40) issued to execution pipe. All instructions have been verified with respect to eight precisions and four IEEE rounding modes, including dynamic rounding specified in floating-point status register. The verification was based on symbolic trajectory evaluation and arithmetic libraries previously proven within the same system [KN02]. The proof includes correctness of the result, update of floating-point status register, and correctness of more than a dozen interrupt signals. Behavior of the floating-point circuitry for invalid instructions and/or instructions with false qualifying predicates have been considered as well. All proofs have been regularly rerun as a regression suite to ensure the consistency of any changes in the design. Formal sequential equivalence checking was used to finish the validation of low-level design proving its correctness with respect to the RTL. However this last phase of verification is out of scope of this paper.

In the process of constructing our proofs we found many bugs and issues that required RTL changes. Our work also helped to clarify incomplete and ambiguous parts of our micro-architectural specification, and it contributed to some hardware optimizations. The proofs are automated and portable to other Itanium® micro-processor designs.

The goal of this paper is to describe the scope and results of our work, and to provide some insight into challenges of using formal verification in an industrial environment, where a fine balance between rigorous verification methods and traditional simulation-based methods is crucial for success of the validation. Although the approach we choose is a combination of known techniques already documented in context of the verification of floating-point adders and multipliers, we believe that it has many aspects that might be interesting to researchers in academia as well as validation engineers.

The paper is organized in following way: Next section describes tools and methodology developed for formal verification of floating-point arithmetic at Intel Corporation and specifics of our approach. The core of our work is described in Section 3, where we dive into details of the verification of the most interesting operation - fused multiply-add, and report what has been covered by our proofs. Since debugging of failing proofs is one of the concerns in the use of formal methods, we touch this question in Section 4. Section 5 focuses on benefits of our effort for the design project. We describe our experience with proof management in Section 6. Concluding section contains summary of our work and detailed comparison to related published work.

2 Our Approach to Formal Verification of FP Arithmetic

Intel’s approach to the validation of floating-point arithmetic includes a huge database of corner test-cases and pseudo-random generators for simulation, as well as Intel’s FORTE formal verification tool that combines theorem-proving with model-checking capabilities ³. The methodology described below does not

³ A publicly available version of the tool that can be used for non-commercial purposes can be downloaded from http://www.intel.com/software/products/opensource/
relies on FORTE specifics and can be reproduced using any tool with capability of symbolic trajectory evaluation (STE) and some means of composing results obtained by STE. We believe that formal proofs coupled with traditional pseudo-random and focused simulation is a good way to achieve thorough functional validation. In our project, the formal and simulation based validation teams mutually benefited from their collaboration. However, this is out of the scope of this paper and we will focus on formal verification only.

2.1 FORTE System and STE

The history of formal verification of floating-point arithmetic at Intel has been motivated by two controversial trends: promising results in academia that were followed by proof of concept at Intel Research Lab [OZ+99]; and bugs that escaped to the micro-processor products [Coe96, Fis97]. Today, formal proofs developed for Pentium(R) designs [AJ+00, KA00, KN02, KK03, Sch03] are reused and even put into hands of validation engineers that are not experts on formal methods. These proofs have been done using FORTE – a system built on top of VOSS. In this section, we give a rather informal description of the technology inside the FORTE system, just enough to understand the paper; details can be found in the referred publications. FORTE includes a light-weight theorem prover and a symbolic trajectory evaluation (STE) engine [STE]. The theorem prover is based on a higher-order logic. The interface language for FORTE is FL - a strongly-typed functional language in the ML family [Pau96]. One good property of FL as a specification language is its executability. While creating specifications, we often ran sanity checks. For instance, the translation from the memory format to register-file format was written as specified by the Software Developers Manual [ISA], and then checked whether consequent inverse translations yield consistent values. FL includes Binary Decision Diagrams (BDDs)[Bry86] as first-class objects and STE as a built-in function. For more information we refer the interested reader to the online documentation for the FORTE system and [KA00]. Here we describe the basic mechanisms of STE and the framework in which we work.

STE is a weak form of model-checking where a formal (gate-level) model is subjected to a symbolic simulation. The idea of a symbolic simulator is similar to that of standard simulator but it differs in that symbolic values (besides explicit binary values) are assigned to each signal and these values propagated through the design model. Results of such simulations are formulas for specified signals at specified times.

STE is an enhancement of symbolic simulation where Boolean logic has been extended to a lattice [STE] with X as a bottom (no information) and T as a top element (overconstrained). X is automatically assigned (by the STE simulator) to signals to which no value has been specified. X can be thought of as an unknown value. Its semantics and use are discussed later. Symbolic values are bound to signals at specified times to form signal trajectories. Trajectories that prune possible computations by restricting the values of some signals at specific
times are called antecedents; they can be interpreted as assumptions. Trajectories that specify expected responses of the circuit are called consequents. A specification is written in a form of Boolean expressions that constrain symbolic values in antecedents and consequents. Trajectory evaluation correctness statement $| \models \text{ckt} \ [\text{ant} \implies \text{cons}]$ means: all circuit computations that satisfy antecedent \text{ant} also satisfy consequent \text{cons}. If any of consequent is violated, a STE run (proof) fails and a counterexample can be extracted from this failure. In fact, the failed proof provides all possible counterexamples and the user may select one for debugging purposes. If all consequents hold at every point of the simulation, success is reported by the tool.

2.2 Pre- and Post-condition Framework

Because of capacity limitations inherit in the STE engine, we may be forced to break our model into smaller pieces. In this case, we make sure that those pieces perfectly fit together. Informally, this means that the border signals of the decomposition match exactly and that nothing is left out of the design. Also the times at which we extract the values of the signals must be consistent. In terms of STE, consequents that include border signals serve as antecedents in the following step of the proof. In this way, we can use facts proved in one part as assumptions for later proofs.

The idea of proof (de)composition described above comes from the pre-and-post-condition theory used for verification of sequential programs. It was first applied to STE by Kaivola and Aagaard [KA00]. It allows one to prove the statements of the form \{P\}S\{Q\}, where \(P\) and \(Q\) are logical properties and \(S\) is a program. In our case, the program is replaced by a circuit and trajectories that bind values inputs and outputs of the circuit at specific times. \{P(x)\}(\text{pretr}_x, \text{ckt}, \text{posttr}_y)\{Q(x, y)\} represents the statement: if \(\text{pretr}_x\) binds the Boolean vector \(x\) to signals (usually inputs) of the circuit \(\text{ckt}\) and \(\text{posttr}_y\) binds the Boolean vector \(y\) to signals of the circuit (usually outputs), then the property \(P(x)\) guarantees property \(Q(x, y)\).

\{P(x)\}(\text{pretr}_x, \text{ckt}, \text{posttr}_y)\{Q(x, y)\} is a shorthand for the following formula:

$$\forall x(P(x) \Rightarrow (\exists y([| \models \text{ckt} \ [\text{pretr}_x \implies \text{posttr}_y]) \land (\forall y(([| \models \text{ckt} \ [\text{pretr}_x \implies \text{posttr}_y]) \Rightarrow Q(x, y)))))) \quad (1)$$

In our methodology, \(P\) is a conjunction of an initial condition that describes the restriction of inputs to the circuit, and an auxiliary pre-condition that is used to further restrict the simulation. For consistency, we use the same initial conditions throughout all proofs for every instruction analyzed, except when we weaken an initial condition to true. An example of an initial input condition is a statement that the specified input signals have value of a specific opcode. Auxiliary pre-conditions are usually used to simplify a particular STE run by restricting symbolic values (meaning that the inputs or internal nodes are restricted). An example of an auxiliary pre-condition is a restriction specifying
a case in a case split. Another example of an auxiliary pre-condition is a side-condition (that we prove separately) used by architects to simplify the design. We refer to $Q$ as the post-condition. Further, $\text{pretr}$ is a union of the initial trajectory that binds symbolic values to the input signals, and the pre-trajectory that binds symbolic values to internal signals. $\text{posttr}$ is referred to as a post-trajectory; it binds symbolic values to signals that we consider as outputs for the purpose of a specific proof.

Intel’s proof libraries contain reasoning rules that apply to STE runs [KA00]. Here we mention those rules that were relevant to our proofs:

- **Pre-condition strengthening**

$$\{P'(x)\}(\text{pretr}_x, \text{ckt}, \text{posttr}_y)\{Q(x, y)\}, \forall x(P(x) \Rightarrow P'(x))$$

- **Post-condition weakening**

$$\{P(x)\}(\text{pretr}_x, \text{ckt}, \text{posttr}_y)\{Q'(x, y)\}, \forall x \forall y(Q'(x, y) \Rightarrow Q(x, y))$$

- **Post-condition conjunction**

$$\{P(x)\}(\text{pretr}_x, \text{ckt}, \text{posttr}_y)\{Q_1(x, y)\}, \{P(x)\}(\text{pretr}_x, \text{ckt}, \text{posttr}_y)\{Q_2(x, y)\}$$

$$\{P(x)\}(\text{pretr}_x, \text{ckt}, \text{posttr}_y)\{Q_1(x, y) \land Q_2(x, y)\}$$

- **Sequential composition**

$$\{P(x)\}(\text{pretr}_x, \text{ckt}, \text{midtr}_z)\{\lambda x.\lambda z.R(x)\}, \{R(x)\}(\text{midtr}_z, \text{ckt}, \text{posttr}_y)\{Q(x, y)\}$$

$$\{P(x)\}(\text{pretr}_x, \text{ckt}, \text{posttr}_y)\{Q(x, y)\}$$

### 2.3 Managing the Size of BDDs

It is important to note that STE uses *Binary Decision Diagrams* (BDDs) [Bry86] to represent formulas produced in the symbolic simulation of a design. BDDs provide a unique representation of Boolean functions, but the space required to represent a Boolean function can critically depend on the order selected for the (decision) variables. Therefore, we need to carefully order the BDD variables. It is well-known that an inappropriate (or random) variable ordering may result in exponential growth of a BDD with respect to the number of input variables. To determine a good variable ordering, we need to know the functionality of particular part of the design. An automatic ordering mechanism called **dynamic re-ordering** is available but it takes additional time and is more suitable for reachability analysis which is not our case. Our rule of thumb in establishing the variable ordering was to put control variables close to the top; and interleave operands’ variables.

It can be tricky to find a variable ordering that is suitable for representing the specification and the design model simultaneously as the former is written independently from the latter. This can be solved by writing a provably equivalent specification that does not have this problem.
In the process of symbolic simulation we try to avoid building formulas/BDDs that do not contribute to the result – the formulas of the signals in the post-trajectory. This can be done by *node weakening* – assigning X (don’t care values) to some signals. Propagation of X’s through the circuit often reduce the complexity of intermediate formulas. Weakening is a safe and conservative way to reduce the complexity of STE simulations. If a node is weakened by mistake, a X will appear as the value for signals in post-trajectory results which causes a proof failure; thus, this is a sound method. Besides user-guided weakening, FORTE has an automatic weakening mechanism that is triggered by the size of the BDD for some nodes.

### 2.4 Verification Methodology

Our methodology was driven by several factors:

- An unusually early start of the formal verification process: our work started at the same time as our traditional simulation effort, i.e., when first lines of RTL code were written.
- Continuous validation effort: proofs has been kept synchronous with changes in the design;
- Limited resources: engineers with experience in formal verification are scarce in the project development.

We looked for the most effective way to achieve high confidence in the design, balancing between investment (learning new tools, writing new specifications, proof maintenance, etc.) and return (covering functionality that cannot be covered by traditional simulation methods with comparable person/time resources). We wished to make a maximal re-use of the formal proofs, and we wanted modularity for an easy maintainability. Our methodology builds on the experience from other Intel groups [KA00, KN02, KK03]. Arithmetic, floating-point, binary and STE proof libraries created for other Intel projects was an important contributer to our success.

Each proof started with a top-down decomposition of the high-level problem into sub-problems, where sub-problems were mapped to bit-level properties checked by FORTE. Decomposition, if needed, was justified by *STE pre-post condition inference rules* [KA00]. However, some simpler instructions did not require decomposition. The gap between the high-level and bit-level specifications was bridged by proof libraries that include IEEE rounding modes [IEEE] and basic floating-point operations like addition and multiplication [KA00, KN02]. These libraries have a clean separation between floating-point values and their encodings that allows customization to particular architectures and micro-architectures. The use of libraries allowed to redirect our focus on writing bit-level specification, describing a mapping from RTL signals and time to mathematical entities, creating environment for debugging and counter-example generation, and overall proof maintenance.
3 Verification of Fused Multiply-Add

3.1 Floating-Point Multiply-Add

The Itanium\(^{(R)}\) ISA defines a floating-point architecture that is fully IEEE compliant for the single, double and double-extended data types, with exponent width 8,11,15 or 17 bits (see [ISA] for details).

Floating-Point Registers (FR) in IA-64 architecture are 82 bits long: The significand field (mantissa) is composed of an explicit integer part (significand\(\{63\}\)) and 63 bits of fraction (significand\(\{62:0\}\)). A 17-bit exponent field defines the magnitude of the number. The exponent is biased. The extreme values of the exponent (all ones and all zeros) are used to encode special values (IEEE Signed Infinity, NaNs, IEEE Signed Zeros [IEEE], the double-extended Real Denormals and double-extended Real Pseudo-Denormals [ISA]). The sign bit indicates whether the number is negative (sign=1) or positive (sign=0).

The value of a finite FP number encoded with non-zero exponent can be calculated from the expression

\[
(-1)^{\text{sign}} \times 2^{(\text{exponent} - \text{bias})} \times \text{significand}\{63\} \times \text{significand}\{62 : 0\} \times 2
\]

where \(\text{significand}\{62 : 0\} \times 2\) denote values represented by a significand\(\{62 : 0\}\) with respect to unsigned binary encodings.

In this paper, we focus on operations applied to normalized operands. Normalized FP numbers have exponents in the range from 1 to 0xFFFFE, and their integer bit is 1. Operations on special values have been covered in our proofs, too. Operations on denormals in considered implementation are deferred to software assist handlers, and our proof obligations consist of raising software assist faults if there is no higher fault.

The floating-point status register (FPSR) is an important element of the architectural state. It contains dynamic control (disabled traps, rounding mode, precision mode, wide-range exponent mode, flush-to-zero mode) and status information indicating traps and faults caused by the the execution of the floating-point operations.

The Floating-Point Multiply-Add (FMA) instruction is one of the most complex IA-64 FP instructions implemented in hardware. One of its important applications is the computation of sums of real and complex matrix products [Nie03]. It is also a basic instruction used for the implementation of division and square root in Intel libraries. The format of \textit{fma} instruction is:

\[
(qp) \quad \text{fma.pc.sff} f1 = f3, f4, f2
\]

The specification dictates that the product of floating-point register (FR) \(f3\) and FR \(f4\) is computed to infinite precision and then FR \(f2\) is added to this product, again in infinite precision. The resulting value is then rounded to the precision indicated by \(pc\) (and possibly FPSR controls) using the rounding mode specified by FPSR. The rounded and normalized result is placed in FR \(f1\). \(qp\)
is instruction qualifying predicate. If \( \text{qp} \) is \( \text{false} \), the instruction has no effect on architectural state.

Considering the range of input values for \( \text{fma} \) instruction (three times 82 bits for operands, 8 precisions, and 4 rounding modes), its validation using exhaustive simulation would require \( 2^{251} \) test patterns to, even disregarding additional control flags, e.g., flush-to-zero.

Hardware implementation of the fused-multiply instruction requires a combination of 64-bit multiplier, an alignment shifter, a 128-bit adder, and rounding logic that includes a leading-zero anticipator and a normalization shifter (see Fig. 1). Implementation details are irrelevant for the purpose of the paper and are left out for confidentiality of the design.

After careful proof planning and manual decomposition of the proof goal to low level properties, our main focus became symbolic trajectory evaluation (STE) of those properties and propositional reasoning. All specifications are written in functional language FL.

We will describe the methodology and how we used it to find bugs in the design of an Itanium\(^{(R)} \) microprocessor. We will mention the problems associated with using formal verification to verify complex data-intensive instructions.

---

**Fig. 1.** Floating-Point Fused Multiply-Add Unit
Several practical issues occur when verifying designs in industrial environment:

- Limited time and human resources;
- Incompleteness of the register-transfer level design;
- Incompleteness of the micro-architectural specification;
- Maintenance of the formal specifications and proofs through the life of the project;
- Need for an efficient regression suite; and
- Complexity of the modern design, which includes features like power management, hardware sharing, scan logic, reset logic, and various levels of abstraction.

In addition to FP exceptions described in IEEE Standard [IEEE], Itanium(R) has a number of faults and traps described in Software Developers Manual [ISA], and many microarchitecture specific interrupts that are described in an Intel internal micro-architectural-level project specification. Besides that, we have to deal with additional complexity due to power management logic, scan logic and reset logic.

Using formal verification in the early phase of the design process means also dealing with the problems related to instability of the model. At the same time, the sooner gets FV involved in the verification process, the more substantial is its impact on finding and fixing bugs right at the beginning. Most problems encountered were found in rounding and the sticky bit computations, and some were a result of incorrect control logic.

Symbolic simulation of all of these constructs in one run is clearly beyond the capacity of today's BDD-based or SAT-based tools. We split the verification task into two subtasks: the correctness of the multiplier and the correctness of adder,rounder and normalizer. Although both tasks have been previously studied, each new design variation provides new verification challenges. In particular, complex power management, scan-out tests, and protection from soft-errors are so much interwind with the main functionality, that the mapping of the mathematical entities to concrete signals at concrete times can be nontrivial. Furthermore, designers use all their experience accumulated during the long years of designing high-performance arithmetic units, e.g. pre-computation of some values, redundancy, taking guess on values and make adjustments later, etc. All these tricks might cause unwelcome surprises to a verification engineer, especially when designer does not provide any mathematical argument for their validity.

Both tasks had to be further decomposed to fit into limitations of the STE engine. Verification of the Multiplier was the harder of the two sub-tasks. The reason for it is that although the algorithm specification gives some indication of further decomposition, it had required a lot of experimentation with symbolic simulation of the model until we found a well-balanced decomposition – a decomposition that was robust to tolerate frequent design changes and fine enough to keep the size of BDDs manageable.