Congresso Brasileiro de Software: Teoria e Prática
28 de setembro a 03 de outubro de 2014 - Maceió/AL

XVII SIMPÓSIO BRASILEIRO DE MÉTODOS FORMAIS
SBMF 2014

Anais
Anais

SBMF 2014
XVII Simpósio Brasileiro de Métodos Formais

COORDENADORES DO COMITÊ DE PROGRAMA
Christiano Braga - Universidade Federal Fluminense (UFF)
Narciso Martí-Oliet - Universidad Complutense de Madrid, Spain

COORDENAÇÃO DO CBSOFT 2014
Baldoino Fonseca - Universidade Federal de Alagoas (UFAL)
Leandro Dias da Silva - Universidade Federal de Alagoas (UFAL)
Márrio Ribeiro - Universidade Federal de Alagoas (UFAL)

REALIZAÇÃO
Universidade Federal de Alagoas (UFAL)
Instituto de Computação (IC/UFAL)

PROMOÇÃO
Sociedade Brasileira de Computação (SBC)

PATROCÍNIO
CAPES, CNPq, INES, Google

APOIO
Instituto Federal de Alagoas, Aloo Telecom, Springer, Secretaria de Estado do Turismo - AL, Maceió Convention & Visitors Bureau, Centro Universitário CESMAC e Mix Cópia
PROCEEDINGS

SBMF 2014
17th Brazilian Symposium on Formal Methods

PROGRAM CHAIRS
Christiano Braga - Universidade Federal Fluminense (UFF)
Narciso Martí-Oliet - Universidad Complutense de Madrid, Spain

CBSOFT 2014 GENERAL CHAIRS
Baldoino Fonseca - Universidade Federal de Alagoas (UFAL)
Leandro Dias da Silva - Universidade Federal de Alagoas (UFAL)
Márcio Ribeiro - Universidade Federal de Alagoas (UFAL)

ORGANIZATION
Universidade Federal de Alagoas (UFAL)
Instituto de Computação (IC/UFAL)

PROMOTION
Sociedade Brasileira de Computação (SBC)

SPONSORS
CAPES, CNPq, INES, Google

SUPPORT
Instituto Federal de Alagoas, Aloo Telecom, Springer, Secretaria de Estado do Turismo - AL, Maceió Convention & Visitors Bureau, Centro Universitário CESMAC and Mix Cópia
Autorizo a reprodução parcial ou total desta obra, para fins acadêmicos, desde que citada a fonte.
Foreword

This volume contains the papers presented at SBMF 2014: the 17th Brazilian Symposium on Formal Methods. The conference was held in Maceió, Brazil, as part of CBSoft2014, the 5th Brazilian Conference on Software: Theory and Practice.

The conference program included two invited talks, given by David Deharbe (UFRN, Brazil) and Narciso Martí-Oliet (Universidad Complutense de Madrid, Spain).

A total of 14 research papers were presented at the conference: 10 full papers and 4 short papers. The abstracts of all papers together with the short papers are included in these proceedings. The authors of full papers will have a chance to revise their papers once more and incorporate feedback received during the conference. Further revised full papers will be published after the conference as a volume of Lecture Notes in Computer Science, by Springer.

The contributions were selected from 34 submissions that came from 18 different countries: Brazil, Canada, Colombia, Denmark, France, Germany, India, Israel, Italy, Pakistan, Portugal, South Africa, Switzerland, Tunisia, Turkey, Ukraine, UK, and Uruguay.

The processes of submission by the authors, paper review and deliberations of the program committee were all assisted by EasyChair.

We are grateful to the program committee and to the referees for their hard work in evaluating submissions and suggesting improvements. SBMF 2014 was organized by the Universidade Federal de Alagoas (UFAL) and promoted by the Brazilian Computer Society (SBC). We are very thankful to the local organizers of CBSoft 2014, that were coordinated by Marcio Ribeiro, Baldoino Santos Neto and Leandro Dias da Silva, all from UFAL, for their very hard work.

We hope you enjoy reading these proceedings as much as we enjoyed preparing them.

September 2014

Christiano Braga and Narciso Martí-Oliet
Program Chairs
SBMF 2014
Palestras Convidadas / Invited Keynotes

b2llvm: B developments onto the LLVM

David Deharbe
We present BLLVM, a multi-platform code generator for the B-method. BLLVM currently handles the following elements of the B language: simple data types, imperative instructions and component compositions. In particular, this paper describes the translation from some essential constructs of the B language for implementations towards LLVM source code. We use an example-based approach for this description.

Equational abstractions in rewriting logic and Maude

Narciso Martí-Oliet
(joint work with J. Meseguer, M. Palomino, F. Durán and A. Verdejo)

Maude is a high-level language and high-performance system supporting both equational and rewriting computation for a wide range of applications. Maude also provides a model checker for linear temporal logic. This procedure can be used to prove properties when the set of states reachable from an initial state in a system is finite; when this is not the case, it may be possible to use an equational abstraction technique for reducing the size of the state space. Abstraction reduces the problem of whether an infinite state system satisfies a temporal logic property to model checking that property on a finite state abstract version. The most common abstractions are quotients of the original system. We present a simple method of defining quotient abstractions by means of equations collapsing the set of states. Our method yields the minimal quotient system together with a set of proof obligations that guarantee its executability and can be discharged with tools such as those available in the Maude formal environment. The proposed method will be illustrated in a couple of detailed examples.
Palestrantes / Keynotes

David Deharbe (Universidade Federal do Rio Grande do Norte)
Dr. David Deharbe is associate professor at Federal University of Rio Grande do Norte (Natal, Brazil) where he has been since 1997. He is a member of the research group ForAll (Formal Methods and Languages Laboratory) and CNPq has granted him a research productivity grant. He is an associate member of VeriDis team, collocated at INRIA (Nancy, France) and Max-Planck Institute (Saarbrücken, Germany). David Deharbe holds a Doctorate degree in Computer Science from Université de Grenoble (France). He did a post-doc at Carnegie Mellon University (Pittsburgh, USA), and two sabbatical at LORIA (Nancy, France). His areas of interest are formal methods (primarily the B-method) and formal verification techniques (first symbolic model checking and then satisfiability modulo theory). In 2014, he served the program committee of ABZ, ICTAC, iFM, SBMF, and co-organized SMT-COMP. He is one of the main developers of the SMT-solver veriT.

Narciso Martí-Oliet (Universidad Complutense de Madrid)
Dr. Narciso Martí-Oliet is a full professor in the Computer Science Department of Universidad Complutense de Madrid (UCM) in Madrid, Spain. He obtained his PhD in Mathematics-Computer Science in 1991 from UCM with a thesis supervised by J. Meseguer on the categorical semantics of linear logic and order-sorted algebra. His research took place from 1988 at the research center SRI International, in Menlo Park, CA, USA, where he was also a postdoctoral international fellow until his return to Madrid in 1994. In 1995 Marti-Oliet took a permanent position as an associate professor in the Computer Science Department of UCM. Since then his research has been focused on the subjects of declarative multiparadigm programming, and algebraic and logical methods for software specification, design and verification, and in particular on several aspects around the use of rewriting logic in such research areas. Marti Oliet currently leads the research group on Formal Analysis and Design of Software Systems (FADOSS) at UCM, where he has supervised four PhD theses. He is principal investigator of several projects of the Spanish Ministry for Science and Innovation and of the Madrid Regional Government, in collaboration with other research groups from Universidad Politecnica de Madrid and UCM. He is also a member of the international team led by J. Meseguer that designs and develops the language and system Maude, based on equational and rewriting logic. An important contribution has been the book "All About Maude, A High-Performance Logical Framework" published in 2007 in the Springer series Lecture Notes in Computer Science. Marti-Oliet has been a member of several program committees for international conferences, and has also organized conferences such as WRLA’04 on Rewriting Logic and its Applications in Barcelona, Spain, RULE’05 on Rule Based Programming in Nara, Japan, and WADT 2012 on Algebraic Development Techniques in Salamanca, Spain. Moreover, since 2008 he is the vicedean in charge of graduate studies at the Computer Science Department of UCM.
Comitês Técnicos / Program Committee

Comitê do programa / Program Committee
Aline Andrade - Universidade Federal da Bahia (UFBA)
Wilkerson Andrade - Universidade Federal de Campina Grande (UFCG)
Luis Barbosa - Universidade do Minho, Portugal
Christiano Braga - Universidade Federal Fluminense (UFF)
Michael Butler - University of Southampton, United Kingdom
Ana Cavalcanti - University of York, United Kingdom
Marcio Cornelio - Universidade Federal de Pernambuco (UFPE)
Andrea Corradini - University of Pisa, Italy
Jim Davies - University of Oxford, United Kingdom
David Deharbe - Universidade Federal do Rio Grande do Norte (UFRN)
Ewen Denney - RIACS/NASA, United States of America
Clare Dixon - University of Liverpool, United Kingdom
Jorge Figueiredo - Universidade Federal de Campina Grande (UFCG)
Marcelo Frias - Instituto Tecnologico de Buenos Aires, Argentina
Rohit Gheyi - Universidade Federal de Campina Grande (UFCG)
Juliano Iyoda - Universidade Federal de Pernambuco (UFPE)
Zhiming Liu - Birmingham City University, United Kingdom
Patricia Machado - Universidade Federal de Campina Grande (UFCG)
Tiago Massoni - Universidade Federal de Campina Grande (UFCG)
Ana Melo - Universidade de São Paulo (USP)
Alvaro Moreira - Universidade Federal do Rio Grande do Sul (UFRGS)
Ana Maria Moreira - Universidade Federal do Rio Grande do Norte (UFRN)
Carroll Morgan - University of New South Wales, Australia
Arnaldo Moura - Universidade Estadual de Campinas (UNICAMP)
Leonardo Moura - Microsoft Research, United States of America
Narciso Marti-Oliet - Universidad Complutense de Madrid, Spain
Alexandre Mota - Universidade Federal de Pernambuco (UFPE)
David Naumann Stevens - Institute of Technology, United States of America
Daltro Nunes - Universidade Federal do Rio Grande do Sul (UFRGS)
Jose Oliveira - Universidade do Minho, Portugal
Marcel Oliveira - Universidade Federal do Rio Grande do Norte (UFRN)
Peter Olveczky - University of Oslo, Norway
Alberto Pardo - Universidad de la Republica, Uruguay
Alexandre Petrenko - CRIM, Canada
Leila Ribeiro - Universidade Federal do Rio Grande do Sul (UFRGS)
Augusto Sampaio - Universidade Federal de Pernambuco (UFPE)
Leila Silva - Universidade Federal de Sergipe (UFS)
Adenilso Simao - Universidade de São Paulo (ICMC/USP)
Heike Wehrheim - University of Paderborn, Germany
Jim Woodcock - University of York, United Kingdom
Comitê organizador / Organizing Committee

COORDENAÇÃO GERAL
Baldoino Fonseca - Universidade Federal de Alagoas (UFAL)
Leandro Dias da Silva - Universidade Federal de Alagoas (UFAL)
Márcio Ribeiro - Universidade Federal de Alagoas (UFAL)

COMITÊ LOCAL
Adilson Santos - Centro Universitário Cesmac (CESMAC)
Elvys Soares - Instituto Federal de Alagoas (IFAL)
Francisco Dalton Barbosa Dias - Universidade Federal de Alagoas (UFAL)

COORDENADORES DO COMITÊ DE PROGRAMA DO SBMF 2014
Christiano Braga - Universidade Federal Fluminense (UFF)
Narciso Martí-Oliet - Universidad Complutense de Madrid, Spain
Índice / Table of Contents

Artigos completos

Parameterisation of Three-Valued Abstractions 12
Nils Timm and Stefan Gruner

A Probabilistic Model Checking Analysis of a Realistic Vehicular Networks Mobility Model 13
Bruno Ferreira, Fernando Braz and Sérgio Campos

Towards completeness in Bounded Model Checking through Automatic Recursion Depth Detection 14
Grigory Fedyukovich and Natasha Sharygina

Completeness and decidability results for hybrid(ised) logics 15
Renato Neves, Manuel A. Martins and Luis Barbosa

A conductive animation of Turing Machines 16
Alberto Ciaffaglione

Mechanised Semantics of BSP Routines with Subgroup Synchronisation 17
Frédéric Gava and Jean Fortin

Formalization of Z-Syntax to reason about Molecular Pathways in HOL4 18
Sohaib Ahmad, Osman Hasan, Umair Siddique and Sofiene Tahar

Test Case Selection Criteria for Symbolic Models of Real-Time Systems 19
Diego Almeida, Alan Moraes, Wilkerson Andrade and Patricia Machado

A generic construction of dynamic logics 20
Alexandre Madeira, Renato Neves, Manuel A. Martins and Luis Barbosa

Model-Driven Engineering in the Heterogeneous Tool Set 21
Daniel Calegari, Till Mossakowski and Nora Szasz

Artigos resumidos

Use Case Analysis based on Formal Methods: An Empirical Study 22
Marcos Antonio de Oliveira Junior, Leila Ribeiro, Erika Cota, Lucio Mauro Duarte, Ingrid Nunes and Filipe Reis
A Proposal for Integrating Formal Methods into a Lightweight UML-driven Development Process  
Thiago C. de Sousa and Paulo Sérgio Muniz Silva

Including Running System Implementations in the Simulation of System of Systems Models  
Kenneth Lausdahl, Claus Ballegaard Nielsen and Klaus Kristensen

Purification of Esterel Programs  
Nir Koblenc and Shmuel Tyszberowicz
Three-valued abstraction is an established technique in software model checking. It proceeds by generating an abstract state space model over the values true, false and unknown, where the latter value is used to represent the loss of information due to abstraction. Temporal logic properties can then be evaluated on such three-valued models. In case of an unknown result, the abstraction is iteratively refined, until a level of abstraction is reached where the property of interest can be either proven or refuted. In this paper, we introduce parameterised three-valued model checking. In our new type of abstract models, unknown parts can be either associated with the constant value unknown or with expressions over boolean parameters. Our parameterisation is an alternative way to state that the truth value of certain predicates or transitions is actually not known and that the checked property has to yield the same result under each possible parameter instantiation. A novel feature of our approach is that it allows for establishing logical connections between parameters: While unknown parts in pure three-valued models are never related to each other, our parameterisation approach enables to represent facts like 'a certain pair of transitions has unknown but complementary truth values', or 'the value of a predicate is unknown but remains constant along all states of a certain execution path'. We demonstrate that such facts can be automatically derived from the software system to be verified and that covering these facts in an abstract model can be crucial for the success and efficiency of checking temporal logic properties. Moreover, we introduce a fully automatic software verification framework based on counterexample-guided abstraction refinement and parameterisation.
A Probabilistic Model Checking Analysis of a Realistic Vehicular Networks Mobility Model
Bruno Ferreira, Fernando Braz and Sérgio Campos

Vehicular Ad-Hoc Network (VANET) is a special type of network where its nodes are vehicles that move according to specific patterns. This network is based on wireless communication, presenting new challenges, such as how it will be tested in realistic scenarios. Currently, simulations are widely used, however, they have limitations, such as local minima. Another approach is Model Checking, which has been used in only a few studies, which often overlook mobility, communication and signal propagation models. This work presents a realistic mobility model using Probabilistic Model Checking to describe an overtake scenario involving three vehicles. This model provides guidelines to represent the mobility aspect and can be connected with other models that represent other aspects of the VANET, such as the network itself. Therefore, VANETs can now be tested using methods closer to the reality.
Towards completeness in Bounded Model Checking through Automatic Recursion Depth Detection
Grigory Fedyukovich and Natasha Sharygina

The presence of recursive function calls is a well-known bottleneck in software model checking as they might cause infinite loops and make verification infeasible. This paper proposes a new technique for sound and complete Bounded Model Checking based on detecting depths for all recursive function calls in a program. The algorithm of detection of recursion depth uses over-approximations of function calls. It proceeds in an iterative manner by refining the function over-approximations until the recursion depth is detected or it becomes clear that the recursion depth detection is not feasible. We prove that if the algorithm terminates, it guarantees to detect a recursion depth required for complete program verification. The key advantage of the proposed algorithm is that it is suitable for generation and/or substitution of function summaries by means of Craig Interpolation helpful to speed up consequent verification runs. We implemented the algorithm for automatic detection of recursion depth in our prototype SAT-based model checker and demonstrate its benefits on a number of recursive C programs.
Completeness and decidability results for hybrid(ised) logics
Renato Neves, Manuel A. Martins and Luis Barbosa

Adding to the modal description of transition structures the ability to refer to specific states, hybrid(ised) logics provide an interesting framework for the specification of reconfigurable systems. The qualifier 'hybrid(ised)' refers to a generic method of developing, on top of whatever specification logic is used to model software configurations, the elements of a hybrid language, including nominals and modalities. In this context, the paper shows how a calculus for a hybrid(ised) logic can be generated from a calculus of the base logic and that, moreover, it preserves soundness and completeness. A second contribution establishes that hybridising a decidable logic also gives rise to a decidable hybrid(ised) one. These results pave the way to the development of dedicated proof tools for such logics used in the design of reconfigurable systems.
A conductive animation of Turing Machines
Alberto Ciaffaglione

We adopt corecursion and coinduction to formalize Turing Machines and their operational semantics in the proof assistant Coq. By combining the formal analysis of converging and diverging evaluations, our approach allows us to certify the implementation of the functions computed by concrete Turing Machines. Our effort may be seen as a first step towards the formal development of basic computability theory.
Mechanised Semantics of BSP Routines with Subgroup Synchronisation
Frédéric Gava and Jean Fortin

This paper presents a core language for BSP algorithms with subgroup synchronisation. We give two mechanised semantics for this language using Coq and prove some common properties on the semantics.
The behavioral characterization of biological organisms is a fundamental requirement for both the understanding of the physiological properties and potential drug designs. One of the most widely used approaches in this domain is molecular pathways, which offers a systematic way to represent and analyze complex biological systems. Traditionally, such pathways are analyzed using paper-and-pencil based proofs and simulations. However, these methods cannot ascertain accurate analysis, which is a serious drawback for safety-critical applications (e.g., analysis of cancer cells and cerebral malarial network). In order to overcome these limitations, we recently proposed to formally reason about molecular pathways within the sound core of a theorem prover. As a first step towards this direction, we formally expressed three logical operators and four inference rules of Z-Syntax, which is a deduction language for molecular pathways. In the current paper, we extend this formalization by verifying a couple of behavioral properties of Z-Syntax based deduction using the HOL4 theorem prover. This verification not only ensures the correctness of our formalization of Z-Syntax but also facilitates its usage for the formal reasoning about molecular pathways. For illustration purposes, we formally analyze a molecular reaction of the glycolytic pathway leading from D-Glucose to Fructose-1,6-bisphosphate.
Test Case Selection Criteria for Symbolic Models of Real-Time Systems
Diego Almeida, Alan Moraes, Wilkerson Andrade and Patricia Machado

In order to avoid an exhaustive search for all possible test cases that can be obtained from a specification model, usually an infeasible, expensive and unnecessary activity, test case generation is usually guided by test criteria. The goal is to produce a minimal test suite and yet effective to reveal faults. Depending on the testing goals and constraints along with the (specific) type of application considered, different criteria may be chosen. However, this choice is not usually straightforward, since most criteria presented in the literature are general-purpose. In this work, we investigate on a hierarchy of criteria that can be applied for test case generation in the scope of model-based testing of real-time systems. The criteria are implemented and applied to Timed Input-Output Symbolic Transition Systems (TIOSTS) in an empirical study. Results obtained indicate that, depending on the criteria applied, failure detection capability of the generated test suite may vary, but differences are not significant for time failures.
A generic construction of dynamic logics
Alexandre Madeira, Renato Neves, Manuel A. Martins and Luis Barbosa

This paper introduces a method to construct dynamic logics with a graded semantics. The construction is parametrized by a structure to support both the spaces of truth and of the domain of computations. Possible instantiations of the method range from classical (assertional) dynamic logic to less common graded logics suitable to deal with programs whose transitional semantics exhibits fuzzy or weighted behaviour. This leads to the systematic derivation of program logics tailored to specific program classes.
When dealing with formal verification, it is sometimes useful to use different formalisms for the verification of each part of the whole problem, as well as to allow formal experts to choose the domain in which they are more skilled to address a formal proof. In this context we have defined a unified environment that allows formal verification within the Model-Driven Engineering (MDE) paradigm using heterogeneous verification approaches. The environment is based on the Theory of Institutions, which provides a sound basis for representing MDE elements and a way for specifying translations from these elements to other logical domains used for verification.

In this paper we present how this environment can be supported in practice by the Heterogeneous Tool Set (Hets). Hets supports heterogeneous specifications and provides capabilities for monitoring their overall correctness. We define semantic-preserving translations from the MDE elements to the core language of Hets, and we also show how it is possible to move from it to other logics, both to supplement the original specification with other verification properties and to perform a heterogeneous verification.
Use Case Analysis based on Formal Methods: An Empirical Study

Marcos Oliveira Junior, Leila Ribeiro, Érika Cota, Lucio Mauro Duarte, Ingrid Nunes, and Filipe Reis

PPGC - Institute of Informatics – Federal University of Rio Grande do Sul (UFRGS)
PO Box 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
{marcos.oliveira,leila,erika,lmduarte,ingridnunes,freis}@inf.ufrgs.br

Abstract. Use Cases (UC) are a popular way of describing system behavior and UC quality impacts the overall system quality. However, they are presented in natural language, which is usually the cause of issues related to imprecision, ambiguity, and incompleteness. We present the results of an empirical study on the formalization of UCs as Graph Transformation models (GTs) with the goal of running tool-supported analyses on them and revealing possible errors. To evaluate our approach, we apply it to a set of real UC descriptions obtained from a software developer company and measured the results through metrics. The final results demonstrate that this approach can reveal real problems that could otherwise go undetected and, thus, help improve the quality of the UCs.

Keywords. Use Cases, Graph Transformation, Model Analysis.

1 Introduction

Use Cases (UC) [2] are a popular model for documenting software expected behaviour. In current practice, UC descriptions are typically informally documented using, in most cases, natural language in a predefined structure. Being informal descriptions, UCs might be ambiguous and imprecise. Thus, the verification of UCs normally corresponds to manual inspections and walkthroughs [4], and detecting problems is not a trivial task. Since software quality is highly dependent on the quality of the specification, cost-effective strategies to decrease the number of errors in UCs are crucial. Strategies for the formalization of UCs have already been proposed, however, many of them assume a particular syntax for UC description tailored for their particular formalisms. This limits the expression of requirements and, in some cases, also restrains the semantics of the UC. Our aim is to keep the expressiveness of a description in natural language and use a formalism for modeling/analysing UCs that is flexible enough to maintain the semantics defined by stakeholders.

In this paper, we investigate the suitability of Graph Transformation (GT) as a formal model to describe and analyze UCs. Some reasons for choosing GT are: the elements of a UC can be naturally represented as graphs; it is a visual
language; the semantics is very simple yet expressive; GT is data-driven; there are various static and dynamic analysis techniques available for GT, as well as tools to support them (e.g., [10]). We work towards an approach that integrates UC formalization and tool-supported analysis, with the objective of improving the quality of UCs. We applied our approach on a set of real UC descriptions obtained from a software development company and measured the results.

This paper is organized as follows: Section 2 presents the necessary background information and an overview of the translation of UCs for GTs.; Section 3 presents the settings of the conducted empirical study; Section 4 presents an analysis and discussion of results; Section 5 presents an analysis of threats to the study; Section 6 presents a comparative analysis of our technique with some related work; and Section 7 concludes the paper and discusses future work.

2 Modeling UCs using GTs

2.1 Background

Use Cases. According to Cockburn (2000) [2], a Use Case (UC) defines a contract between stakeholders of a system, describing part of the system behavior. The main purpose of a UC description is the documentation of the expected system behavior so as to ease the communication between stakeholders, often including non-technical people, about required system functionalities. For this reason, UC descriptions are usually described in a textual form.

Graph Transformations. The formalism of Graph Transformations (GT) [7] is based on defining states of a system as graphs and state changes as rules that transform these graphs. Our analysis of GTs is based on concurrent rules and critical pairs, two methods of analysis independent from the initial state of the system and, thus, they are complementary to any other verification strategy based on initial states (such as testing).

2.2 Proposed Formalization and Verification Approach

The proposed approach, detailed in [6], takes as input a textual UC description, from which the entities and actions that will be part of the formal model are identified. Then, basic verifications can be performed regarding the consistency of the extracted information. If inconsistencies are detected, the UC must be rewritten to eliminate them or the analyst can annotate the problem to be resolved later on. When no basic inconsistencies are found, the GT can then be generated. In this process, conditions and effects of actions are modeled as states and a type graph is built. After that, each UC step is modeled as a transition rule from one state (graph) to another. Having the GT, a series of automatic verifications can be performed to detect possible problems.

We use the AGG tool [10] to perform the automatic analyses on the GT model. All detected issues are annotated as open issues (OIs) along with the solutions (when applicable). Open issues are classified according to their severity.
level: Yellow (for warnings), Orange (for relevant issues), or Red (for critical issues). The actions to be taken regarding found OIs depend on the analysts, who can determine whether an OI is in fact a real problem.

3 Empirical Study Settings

In order to adequately evaluate our approach, we followed the principles of Experimental Software Engineering [11] and the GQM template [1]. Our main study goal was to demonstrate the usefulness of GTs to improve the quality of UCs by the identification of OIs, from a perspective of the researcher, in the context of a single real software development project. From this, we derived two research questions, which we aimed to answer with our study.

RQ-1 Are system analysts able to detect problems in their own UC descriptions without additional support?

RQ-2 How effective is our GT-based approach in identifying problems in UCs?

The UC descriptions we used in our study are part of the analysis documentation of an industrial software project. This project involves the development of a typical system to manage products from a warehouse, with functional requirements such as adding new products, creating sale orders, and releasing products in stock. We do not provide any further details about our target system and its UCs due to a confidentiality agreement.

3.1 Procedure

The procedure of the study consists of the following steps:

1. Analysis by System Analyst. We requested a system analyst responsible for the UC descriptions to carefully revise them, and point out problems, such as ambiguity, imprecision, omission, incompleteness, and inconsistency.

2. UC Formalization. Given a set of UCs, we performed the steps detailed in [6] to formalize them using GTs and used the AGG tool to analyze them, detecting some OIs.

3. Evaluation of Open Issues. After identifying OIs, we had evaluated whether detected OIs were real problems in the analyzed UCs.

4. Data Analysis. Our aim is that our approach detects all and only real problems. This can be seen as a classification problem, and thus the effectiveness of our approach can be measured using the metrics, widely used in the context of information retrieval, of precision and relative recall [5], whose formulas are shown below, where true positives are OIs that correspond to real problems; false positives are OIs that are not real problems; and false negatives are real problems not identified as OIs.

\[
\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} \tag{1}
\]

\[
\text{Relative Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \tag{2}
\]
4 Results and Discussion

After revising the original UCs, the system analyst found no problems. However, after applying our approach to these UCs, we identified 32 OIs across the 5 UCs, which gives an average of 6.4 OIs per UC. This is an expressive number, given that the system analyst stated that the UCs had been correctly specified. In order to verify whether the identified OIs were false alarms (false positives), the system analyst was asked to check each one of them. Of the 32 OIs, 24 were pointed out as real problems and only 8 as false positives.

Table 1 presents our results in detail. It shows the number of OIs found in each UC (columns labeled with OI) and how many of these OIs were confirmed as real problems (columns labeled with P). The rows show the number of detected OIs with respect to their level of severity. The table also presents the total number of detected OIs and the total number of real problems considering all the 5 UCs. The symbols ▲, ○, and ◊ represent warnings (severity Yellow), relevant issues (severity Orange), and critical issues (severity Red), respectively.

We then analyzed these results according to the selected metrics. Because the system analyst was unable to identify any problem without support, the number of problems not identified by our approach was 0, leading to relative recall = 1.0. As for the Precision, we obtained 0.75 (24 true positives and 8 false positives) — that is, 75% of the OIs identified by our GT-based approach were real problems. Not only most of the identified issues were actual problems, but also most of the false alarms (7 of 8) were related to low severity OIs.

By analyzing OIs not identified as problems, we observed that 6 of them were not necessarily classified as a false positive by the system analyst. They preferred to leave such issues as they were and postpone changes to future design decisions, considering that they alone could not decide what was the best approach to tackle those issues. The other 2 OIs found, confirmed as false positives, were related to incompleteness or ambiguities due to the lack of knowledge of the modeler about the problem domain and the internal processes of the company.

Note that OIs were identified without the intervention of any stakeholder. The only provided input was the software documentation in the form of UC descriptions and the output was a checklist with OIs to be revised. More importantly, had these problems been detected before the design and implementation, when they should have, development costs could have been potentially reduced.
5 Threats to Validity

During our study we carefully considered validity concerns. This section discusses the main threats we identified to validate this study and how we mitigated them.

Internal Validity. The main threat to internal validity of this study was the selection of modeler of UCs in the formalism of graphs. However, we want to show that, correctly following the steps of our strategy, the modeler does not need a deep understanding of the formalism. Moreover, we used the AGG tool to automate the analyses and provide a graphical interface.

Construct Validity. There are different ways of modeling a system through the formalism of graphs that can produce some threats to construct validity. The modeler may not follow correctly the modeling steps, being influenced by their prior knowledge about the formalism. Therefore, we proposed a roadmap, step by step, on how to model UCs as GTs, for both beginners and experts users.

Conclusion Validity. As the main threat to validity of the conclusion we highlight potential problems in the generation of the model in the formalism of graphs. Once again, our step-by-step modeling process should be followed to prevent the modeler from creating a model that is not consistent with the textual description. Moreover, the tool-supported verifications can also detect such modeling errors, thus reducing the risk of this threat.

External Validity. The main threat to the external validity was the selection of artifacts on which we based our study. We did not use any criteria to select either the project or the system analyst who participated of our study. We were aware of this threat during the study. However, we opted for randomly choosing artifacts to support the applicability of our strategy in different scenarios.

6 Related Work

Some authors have developed approaches for translating UCs to well-known formalisms, such as LTS [8], Petri Nets [12], and FSM [9]. Unlike these formalisms, a GT model is data-driven and we do not need to explicitly determine the control flow unless it is necessary to guarantee data consistency. The approach presented in [13] allows the simulation of the execution of the system but do not report the use of any type of analysis, which, in our opinion, reduces the advantage of having a formal model. The work described in [3] considers analyses such as critical pairs and dependencies involving multiple UCs and provides some ideas on the interpretation of the results. However, we propose a more structured way of providing diagnostic feedback about single UCs, which serves as a guide to point out the possible errors as well as their severity level.

7 Conclusions and Future Work

We investigated the suitability of GT as a formal basis for UC description and improvement. We evaluated our approach through an experiment with real software artifacts, where we could detect existing errors, which helped improve the
original UCs. Making a general analysis of the experiment, we consider the results promising, since it was possible to identify a large number of real problems based on a documentation that was produced at an early stage of software development. We observed the need for further automating the process, if not all, at least some steps, which is one of the most immediate planned future work.

A inter-UC analysis is currently being implemented as well as a more detailed diagnostic feedback. Within the same model frame, other types of validation and verification techniques on GT models, such as test case generation, model checking, and theorem proving, are also subject of current work. We plan to investigate whether we could reduce the impact and cost of changes by identifying which parts of the description are affected. Finally, note that, although we did not present any new formal method or verification technique here, a considerable amount of expertise in formal methods was required to define the OIs: they are meant to bridge the gap between the informal and formal worlds. We believe that this type of work is crucial towards the industrial adoption of formal methods.

References

A Proposal for Integrating Formal Methods into a Lightweight UML-driven Development Process

Thiago C. de Sousa and Paulo Sérgio Muniz Silva

1 State University of Piauí, Brazil
thiago@uespi.br
2 University of São Paulo, Brazil
paulo.muniz@usp.br

Abstract. The best practices of software engineering indicate that the verification activity is essential to achieve some quality during the software construction. In UML-based development processes, one of their main focuses is the detection of inconsistencies in diagrams that represent the software. However, most of these processes, such as ICONIX, apply only informal techniques (eg. visual model inspection), often implying the negligence of that activity by developers. Moreover, with the advance of automated verification tools, formal methods, such as Event-B, are increasingly attracting the attention of software companies. However, it is still difficult to convince developers to adopt them, because they are not acquainted with some of their mathematical concepts. Thus, this paper presents a proposal for the inclusion of Event-B within ICONIX, giving rise to BICONIX, an object-oriented development process that supports inconsistencies formal verification. Specifically, this work shows how this merger can assist the verification activity in well-defined check points of the proposed process.

Keywords: Formal Verification, UML, ICONIX, Event-B

1 Introduction

UML has become the “de facto” standard for software modeling and, nowadays, there are a lot of development processes that use UML diagrams to create partial models of the system being produced. These models usually describe a system from different viewpoints and levels of abstraction and, frequently, lead to a number of inconsistency issues, which are well-known by the software engineering community. One of these problems is to make sure that all models respect the constraints (business rules and functional properties) imposed by the application domain and/or by the stakeholders. Another issue is how to ensure that each software model has a unique interpretation (precise semantics), which means that it cannot be understood in different ways by two or more developers. Finally, there is the problem of checking whether the semantics of an abstract model is preserved by its detailed versions after one or more successive refinements.

On the one hand, most of software development processes based on UML, such as RUP and ICONIX [1], include the verification of these inconsistencies as
an essential task. However, many verification techniques are based on inspections of the models and, as the UML models and the constraints are expressed by informal languages, the inspections are usually carried out manually and visually, making them costly and strongly dependent on the skill and experience of the developer.

On the other hand, formal methods, such as VDM [2], B [3], Event-B [4] and Z [5], have supported not only to check model inconsistencies precisely and automatically, but also work as software specification and modeling tools. However, despite the effectiveness of formal techniques in filling the gap between requirements specification and implementation and ensuring system correctness by construction (verification mechanism), industrial practitioners are still reluctant to fully adopt them.

As we can realize, although inconsistencies verification is a critical issue in software engineering, the widely used software development processes based on UML have no efficient mechanism to perform this task. It is also observed that formal methods provide effective techniques to address this problem, but they do not attract the attention of the community. So, for many industrial practitioners it would be very useful to have a well-known UML-based process with support for formal verification. In this work we present an approach for the integration of the ICONIX process with the Event-B formal method. More precisely, we show how the Event-B formalism can be incorporated into the ICONIX steps/stages in order to provide a mechanism to check for inconsistencies.

In the next section we present some theoretical fundamentals, introducing the features of the ICONIX process, as well as the main concepts of the Event-B language. In section 3 we explain our approach named BICONIX in more details, showing an structure overview and the main tasks of each phase. In section 4, we show some related work, and finally we reserve the last section for further discussions about how BICONIX can assist the verification task and directions for future work.

2 Background

The ICONIX process can be considered a pure, lightweight and practical, but also powerful methodology. ICONIX is not as bureaucratic as RUP, which means that it does not generate a lot of documentation. And despite being a simple process like XP (eXtreme Programming), it does not lack the Analysis and Design phase, providing a very simple step-by-step guiding rules during the whole process. Moreover, ICONIX uses only four diagrams (use cases, robustness, sequence and class), follows the iterative and incremental development cycle and brings the developer to a “mandatory” and well-defined verification task among its phases in order to check requirements compliance.

Recently, a variant of the B formal method has been successfully applied in some companies from different fields, such as aerospace, services and railways. Event-B is a state-based formal method for modeling systems based on predicate logic and set theory where the refinement mechanism, which allow us to build a
model gradually by making it more and more precise (that is, from an abstract model to a concrete one), and the consistency checking, which allows us to verify the validity of the properties of a system, are guaranteed by mathematical proof obligations. These features have been supported by an open-source platform (based on Eclipse IDE) named Rodin, that is constantly improved and extended by the community via plugins.

3 The BICONIX Process

In this section we present an object-oriented development process named BICONIX that supports formal verification of inconsistencies. The proposed process keeps basically the main characteristics of ICONIX and extends it by incorporating a specific role (the Event-B Profile) for allowing the addition of invariants (business rules and functional properties), guards, actions and refinement relations among the models generated automatically in each of its first three phases. An overview of BICONIX can be seen in Figure 1.

![Fig. 1. Overview of the BICONIX process](image-url)
The BICONIX process has two dimensions: the horizontal axis represents the temporal order and shows the lifecycle aspects of the development process; the vertical axis is used to represent both structural and behavioral software aspects. The first dimension represents the dynamic aspect of the process and is expressed in terms of phases, milestones and iterations. The second dimension is the static aspect of the process, as it is described in terms of models, activities, workflows, artifacts and profiles.

The BICONIX process is purposely very similar to ICONIX in order to invite regular developers to using it. So, BICONIX has also four sequential phases, each one concluded by a milestone. At the end of a phase there is an execution of a critical review in order to determine whether the objectives have been achieved. A positive evaluation allows the project to move forward to the next stage. Basically, the artifacts, activities and workflows of the BICONIX process are the same ones of ICONIX, only including those ones related to the Event-B profile, keeping the lightweight aspect of ICONIX. Due to space limitations, we will not detail BICONIX, only focusing on the differences from ICONIX.

One of the main novelties of BICONIX compared to the original process is the support of formal verification, which appears after the transition between the ICONIX and Event-B profiles. At the end of each of its first three phases, the Event-B specialist receives the diagrams produced by the ICONIX developer and uses the Rodin platform, extended with some transformation rules, for automatic translation of them into the Event-B language. Basically, the artifacts that represent the static part of the software (Domain Model, Updated Domain Model and Class Diagram) generate sets, relations and type invariants in Event-B, and the artifacts that represent the dynamic part of the software (Use Case, Robustness and Sequence Diagrams) are mapped to events.

The generated formal model may be augmented with constraints (invariants) and pre/post-conditions (guards/actions). In the first phase (Requirements Analysis), the Event-B expert can define constraints over the domain model and pre/post-conditions on the use cases. In the second phase (Analysis and Preliminary Design), some constraints over the updated domain model (with attributes) and pre/post-conditions on the robustness diagrams can be described. Finally, at the stage of Detailed Design, the specialist can include constraints over the class model (with methods) and pre/post-conditions on the sequence diagrams.

The final formal model produced is then checked automatically by the tool in order to detect errors. If there are any issues, they are discussed by the two roles (profiles) during the review activity of each phase (requirements review, preliminary design review or critical design review). Essentially, the problems occur due to invariants broken by event actions or due to a detailed model that is not following the formal refinement rules. For mapping back the Event-B model errors as ICONIX artifacts issues, the specialists are aided by an implicit dictionary (e.g. the constraint word in the ICONIX context is synonyms of the invariant word in the context of Event-B). So, the developers must decide which steps of the current phase, including the diagram construction, should be redone in order to correct the errors, before proceeding to the next one.
4 Related Work

Relevant work present formal UML-based methods. Runde et al. [6] present a formal method called STAIRS based on Interactions and Sequence diagrams as defined in UML 2.0, with the horizontal and vertical refinement notions. Ke et al. [7] detail a sound process for developing critical systems named rCOS, which translates some UML diagrams into a formal language and checks them using the FDR tool. Ahrendt et al. [8] present a formal object-oriented process named KeY, which uses UML diagrams and OCL annotations to generate verified code in Java Card format using a tool developed by the authors. However, they are not based on any known methodology, which may not encourage their use.

Other work use our same approach to provide a mapping from UML to a well known formal language. Laleau and Mammar [9] provide a method to generate B specifications from UML (class, state and collaboration diagrams) and check them using the AtelierB tool. Lausdahl et al. [10] present an approach to translate Class and Sequence diagrams to VDM++ and check them using the Overture tool. Miao et al. [11] show how to translate Class, Sequence and Statecharts diagrams into Object-Z and check them using the OZRose tool. But there is no mention to integrate them in a diffused methodology like we do.

5 Discussions and Future Work

In this paper we have proposed an approach for including a formal method (Event-B) into a lightweight UML-based methodology (ICONIX) in order to aid industrial practitioners with the verification task. Since there is the formalization of the artifacts, the user of the BICONIX process has confidence to check both syntactic (eg. every use case must have a name) and semantic (eg. no cycles in a use case diagram) modeling issues accurately.

We believe that the introduction of a formal method in a development process brings, under the technical point of view, two major benefits. The first one is the discovery of modeling problems at the early stages, which contributes significantly to the reduction of rework, essential to minimize the technical debt. A second gain would be the obligation of performing the verification activity in specific check points of the process, since BICONIX forces the execution of this task between the transition of its phases, thus contributing to the dissemination of the formal analysis culture.

From the manager’s viewpoint, it is important to emphasize that the BICONIX process proposes the coexistence of two technical roles to perform its many activities: the ICONIX and the Event-B profiles. At a glance, this feature may be considered an additional issue to manager control, impacting negatively upon the management of projects that follow the process. However, we believe that the inclusion of a specialist that fits the Event-B profile, for managers who already have experience in the ICONIX process, should not be a major issue, since the tasks under the liability of the expert are performed only at the end of each phase, having no impact on the ICONIX process kernel.
Finally, this work requires important future improvements, many of them related to the BICONIX limitations: the inclusion of elements in the Event-B language for supporting the object-oriented semantics; the addition of other standard elements of the UML diagrams (eg, messages types of Sequence Diagram); the definition of refinement patterns to accelerate the description of gluing invariants among the generated Event-B models in each phase; the complementation of the process to enable the addition of invariants and guards/actions in the Implementation phase, providing an integration with a language that allows specifying these details directly in the source code; the integration of an easier constraint language in order to reduce the responsibilities of Event-B specialist; the association of a tool for automatic test generation in order to reinforce the verification mechanism; the incorporation of a mechanism to assist the transition from requirements specification to Use Cases and Domain models, developing, for example, a controlled language to express them; the improvement of the feedback from Event-B errors provided by the Rodin platform to diagrams issues; and the elaboration of a controlled experiment to assess the viability of the proposed process in real projects.

References

Including Running System Implementations in the Simulation of System of Systems Models

Kenneth Lausdahl\textsuperscript{1},
Claus Ballegård Nielsen\textsuperscript{1}, and Klaus Kristensen\textsuperscript{2}

\textsuperscript{1} Department of Engineering, Aarhus University.
Finlandsgade 22, Aarhus N 8200 Denmark
{lausdahl,clausbn}@eng.au.dk
\textsuperscript{2} Bang & Olufsen, Denmark
krt@bang-olufsen.dk

Abstract. The formal modelling of System of Systems is challenged by the autonomy of the participating constituent systems as it may not be possible to obtain the implementation details needed to create a descriptive model. This paper describes an extension for the Symphony tool that enables formal models of System of Systems to be connected and simulated with externally running system implementations. An industrial case study of a Bang & Olufsen sound system is used to show the application and feasibility of the approach.

Keywords: System of Systems, Formal Modelling, External Systems

1 Introduction

A System of Systems (SoS) is a type of system that is composed of largely independent constituent systems, which collaborate in order to reach a common goal [4]. In their nature SoSs consist of distributed constituent systems that have a high degree of autonomy and often are developed and evolved individually. The complex structures and interactions found in an SoS can be described formally, and the challenges faced by system developers can be analysed through formal modelling [3].

The COMPASS Modelling Language (CML) is a formal modelling notation aimed at creating models of SoS architectures and SoS properties in order to help developers analysing SoS [8]. Despite the tool-support and a formalism aimed at describing SoS, the modelling of SoS is however still challenging. The two main causes of this is that: 1) the independent owners of the systems may not be willing to share all knowledge on their system implementation, making it difficult to create models with the correct behaviour, 2) the systems may be legacy systems with no precise description of their internals. In this paper, we show how SoS models can incorporate external constituent systems as part of the model simulation through improved tool-support.

The remainder of this paper is structured as follows: Section 2 gives an overview of the challenges in formal modelling of SoS. The principles of the simulation between the CML simulator and the external system, as well as details of the tool extension are described in Section 3. A small study on using the external simulation in an industrial case is presented in Section 4, before we draw conclusions in Section 5.
2 System of Systems Engineering and Formal Modelling

SoS Engineering is challenged by a strong degree of autonomy, evolution and emergence [4]. In order to address this, the use of formal methods has been proposed as a way of providing better analyses of the system design [1]. The formal languages directly aimed at SoS modelling are however still challenged by the autonomy constituent systems. As there can be conflicting interests between the owners of the individual constituent systems, or because parts of the SoS are delivered by a supplier that has no interest in the SoS, it may be difficult to create sufficiently descriptive formal models. The owners of the independent constituent systems may not be willing to share implementation details for competitive reasons, and suppliers may just have delivered a commercial off-the-shelf product for which they have no interest in delivering documentation on its internals. Finally, some of the constituent systems in the SoS may be legacy systems for which documentation of their system implementation is no longer producible. This creates challenges in the simulation of the executable formal models, as a meaningful simulation depends on the actual behaviour of the systems involved.

We present a tool extension to simulate SoS models for which the behaviour of some of the constituent systems can only be obtained via the actual running system, by establishing connectivity between the simulator and an external system.

Including running systems in the simulation is already possible in other formal methods. The Overture tool has a mechanism that allows VDM models to delegate calls to external Java libraries [5], and for Coloured Petri Nets, CPN Tools offers the Access/CPN functionality which enables the integration of CPN models with external applications via a TCP/IP stream [7]. These approaches do, however, use a different approach that either requires modification of the model or for the external system to control the simulator.

2.1 The COMPASS Modelling Language

CML combines elements from the VDM state-based formal method [2] and the CSP process-based formal method [6] in order to express both the structure and behaviour of an SoS. An SoS is modelled as a collection of constituent systems with their behaviour specified in processes, which have communication channels defined between them.

The CML language is built around collections of types, values, functions, operations, classes, processes and channels, with the types, values, functions and classes originating from VDM, while channels are taken from CSP/Circus. The processes are the central constructs in CML as they act as the constituents in the SoS. A process can define and maintain state using the VDM based type system. Finally, actions are used to express the reactive behaviour of a process. Channels are globally defined and are used for defining the communication and synchronization events between processes.

The Symphony tool\(^3\) is an Eclipse-based IDE that provides a CML parser, type-checker and interpreter & debugger for model simulation, as well as range of static analyses such as a theorem prover, proof obligation generator and model checking.

An example of a very basic CML model is given in Listing 1.1, which shows the definition of a channel \(c\) carrying a nat type, and three small systems represented by the processes \(P\), \(A\) and \(B\). A parallel composition is made between the \(A\) and \(B\) processes.

\(^3\) Symphony Tool webpage: https://github.com/symphonytool/
as defined by the process $P$. A process, such as $P$, that defines the system composition at the highest level is known as the top process. When making a composition, a range of channels can be given to denote the channels that the composed process can synchronize on, based on the events occurring on the channels. In the given example the composition defines that the processes will synchronize on events occurring on channel $c$. Process $A$ and $B$ have a very simple functionality, where $A$ is a system that defines an action (indicated by @) to synchronize the value 1 on channel $c$, while $B$ is a system that synchronizes a value placed on channel $c$ (indicated by ?) and assigns it to $x$.

Within the simulator, a CML model is represented as a tree structure of model behaviours. The top process is used to define the entry point for the simulation, and the simulation is performed by moving from the top and down the tree to inspect and execute underlying behaviours. An inspection will reveal the collection of events that the composed systems can execute. The structure for Listing 1.1 is shown in Figure 1, where the inspection of the composition results in the collection of possible events. A parent in the tree structure functions as a coordinator for its children, meaning that it will filter the collection of events available to the children and thereby control the possible execution.

```plaintext
channels c : nat
process P = A [\{c\}] B

process A = begin
  @ c.1 -> Skip
end

process B = begin
  @ c?x -> Skip
end
```

Listing 1.1: CML example

Fig. 1: Internal simulator representation of $P$ from Listing 1.1.

### 3 Tool Support for Simulation with External Systems

The CML simulator in the Symphony tool has been extended such that systems that are running externally to the simulation environment can be included in the simulation of CML models. This means that concrete system implementations can be run and executed in parallel to the simulation of the model, allowing the behaviour of both the model and the system to affect each other. The presented approach can be divided into three subsections: the principles of the approach (Section 3.1), the changes to the simulator (Section 3.3) and the protocol used to establish the interaction (Section 3.2).

#### 3.1 Principles

When an external running system implementation is going to be part of the simulation of a CML model, it will act as one of the constituent systems defined in the overall SoS. As each constituent system is expected to have its own specified behaviour and have a specified way of communicating with the other constituent systems in the SoS model, it is possible to make the external system act as a constituent system in the model. Essentially, the tool extension will replace a process definition in the model with a skeleton that will delegate the interaction to the external system.
The Symphony tool will be in control of the simulation and will act as a *coordinator* in the simulation, while the external system will act as a client. The coordinator is responsible for starting the simulation and for controlling the flow of the simulation, once started. A special debug configuration in the Symphony tool is used to setup the simulator as a coordinator and for interacting with external systems. In the configuration of the coordinator the top level process is selected, as well as the processes that will be handled by an external system. The debug configuration also requires address and port information to be specified, such that the clients can connect to the coordinator.

During the simulation the coordinator will control the flow of the model execution according to the model specification. The possible transitions that can be made in the model are computed in the same way as for a normal simulator. The only change is that some of the computations are performed by external systems. From a simulation point of view the executing model is essentially seen as one model, which is not distinguishable from a normal simulation.

### 3.2 Protocol

The connectivity and data exchange between the simulator and the externally running system is performed via a custom protocol built on top of a TCP connection. The protocol for handling the simulation reflects the simulator’s way of inspecting and execution behaviours, and the protocol has been formally specified in a CML model. To facilitate the interoperability between the simulator’s Java implementation and the running systems, that may be software implemented with different technologies on different platforms, a data-interchange format is used. The JavaScript Object Notation (JSON) is used, as this will ensure interoperability. The protocol mirrors the execution flow of the normal simulator by providing message types for the inspection and execution of behaviours. The protocol keeps a flow state in order to ensure that an execution message is only allowed if an inspection message has previously been processed. The protocol also contains message types that allow clients to register on the coordinator, as well as messages for disconnecting. The network functions as client–server relationship, with the coordinator functioning as a server that the clients can connect to during the initialisation of the simulation.

### 3.3 Simulation with External System

The Symphony simulator has been extended such that it changes the way it handles behaviours. Essentially, all the processes that have been specified as being handled by an external system in the debug configuration will be replaced with a skeleton that delegates the processing to the external system via a network setup. As the protocol imitates the inspection and execution calls occurring in the normal simulation, the coordinator can use most of the existing simulator implementation and function almost like a normal simulator, just with some calls being delegated over the network.

This is illustrated in Figure 2, that shows: (a) the tree structure of a normal simulator and (b) the same tree structure for simulation where the process $B$ is an external system. The dashed line represents the network connection between the simulators.

The external system needs to have an adaptor between the system implementation and the network setup to the simulator. The adaptor is responsible for connecting to the
simulator over the network and for implementing the simulation protocol specifying the possible interactions (Section 3.2). The implementation of the protocol involves two steps: 1) a mapping of the types of the concrete programming language to the CML types described in the simulation protocol, and 2) a mapping from the protocol messages to operations in the external system itself. Essentially, the external system needs to provide the simulator with a list of transitions that represents the possible events that the system can perform given the current system state, and allow for the simulator to send an execute message that can change the system state. This can be implemented through the use of a state machine in the adaptor, which is used in the case study Section 4.

4 Case Study

The approach was examined through an industrial case study involving a Bang & Olufsen (B&O) home Audio/Video (A/V) network for connecting devices (such as audio, video and legacy audio products) distributed across a user’s home. These devices may be produced by competing manufacturers, but need to interact in order to deliver a service to the user. As such, the A/V network forms an SoS of devices that can both be heterogeneous and legacy systems. The devices may provide stand-alone streaming or media content rendering services, but the SoS needs to deliver a coherent experience. A key part of the system is the A/V control that has the task of managing which of the devices should stream and render media. This behaviour is defined through streaming contracts that specify the interactions between the devices.

A CML model of the A/V network streaming contracts is used to examine the presented approach. The CML model contains 2 devices: (1) the A/V control representing the user’s interaction semantics, and (2) a streaming device which renders media on the basis of control events from the A/V control. The CML model describes the streaming contracts by defining a set of semantically defined transition rules for distributed state synchronization and distributed operation calls.

The externally running system used for this case study is a B&O developed C++ implementation of the streaming contract. The implementation makes use of two interfaces: 1) a call-back interface where application layer clients will be notified of transition operations and states changes, and 2) an interface that functions as an application layer abstraction creating an adaptor towards the platform specific streaming implementations. These two interfaces maps to the channels defined in the CML model of the streaming contract, and the semantics of these interfaces’ implementations satisfy the transition rules defined for the streaming contract. In order to map the externally running system to the events being described in CML, a state machine has been implemented where the channel events of the CML model are translated into operations.
calls in the C++ implementation. For the network setup between the simulator and the external system some glue code in the C++ implementation is used to handle the mapping for states, types and operation logic. This also works as a wrapper that creates the needed JSON messages for the external system.

Being able to include the running system as part of the model simulation, enabled B&O to check the behaviour described in a model against the real system implementation. In the SoS model a process modelling a constituent system was replaced with the actual system being modelled. Testing a concrete system up against a model showed that the behaviour described in the model corresponded to the systems implementation for the given modelled scenario, and vice versa. This same approach can be used to replace parts of the SoS model with existing systems from other manufacturers or with legacy systems.

5 Conclusions

We have presented an approach for including externally running systems in the simulation of formal models of SoSs. By the use of a tool extension it has become possible to include constituent systems that, for various reasons, cannot be modelled into the simulation of a complete SoS model. We have shown the application of the approach through a small industrial case study and have demonstrated its feasibility. Allowing the simulation of SoS partly consisting of running constituent systems enables developers to address the challenges of autonomy and legacy often found in SoS development, while still being able to use a formalism targeted SoS development. We believe that the approach described in this paper can be used as an inspiration for tool builders of other formal methods that have tool–support for performing simulations of models.

Acknowledgment

The work presented here is partly supported by the EU Framework 7 Integrated Project: COMPASS, Grant Agreement 287829. The authors would also like to thank the anonymous reviewers for their valuable comments and suggestions.

References

Purification of ESTEREL Programs *

Nir Koblenz and Shmuel Tyszberowicz

1 Department of Mathematics and Computer Science, Open University of Israel, Raanana, Israel
2 School of Computer Science, Academic College of Tel-Aviv Yaffo, Tel-Aviv, Israel
skoblen@gmail.com, tyshbe@tau.ac.il

Abstract. ESTEREL is a synchronous programming language dedicated to control-dominated reactive systems. XEVE, an ESTEREL verification tool, verifies circuit descriptions generated from the source programs. However, the ESTEREL compiler produces circuits with a behavior equivalent to the original program only for programs that do not handle data. The abstraction performed by the compiler removes the data, yielding an over-approximation that might violate safety properties even when the source program does not, causing XEVE to reject correct programs. We introduce an automatic abstraction process for ESTEREL programs developed to tackle this problem. When applied to a program, it results in a pure program with an equal external observable behavior. When applying the abstraction to a program augmented with observers that monitor safety properties, the ESTEREL compiler can compile the resulting pure program into a verifiable equivalent circuit. We have built a prototype tool that implements the abstraction and used it to purify control programs and robotic systems.

Keywords: Verification, Abstraction, Reactive Systems, Esterel

1 Introduction

Reactive systems are computer systems that continuously react to their environment at a speed determined by the environment. Most industrial real-time systems are reactive [6]. ESTEREL is an imperative concurrent language for the development of industrial-strength reactive systems, which is especially well suited for control-dominated reactive systems such as real-time process control systems, embedded systems, communication protocols, peripheral drivers, human-machine interfaces, and others [1]. ESTEREL belongs to the family of

* The paper is based on the master’s thesis of the first author. A draft of the thesis and the prototype tool are available at http://www.cs.tau.ac.il/~tyshbe/NIR/nirThesisDraft.html. This work has been partially supported by GIF (grant No. 1131-9.6/2011).
1 We refer to ESTEREL v5.92, which we use for teaching. The toolset and the documentation are available at http://www-sop.inria.fr/esterel.org/filesv5_92/.
synchronous languages—languages that are based on the synchrony hypothesis, which states that a program instantaneously reacts to its input. Control is assumed to take no time and thus output is broadcast right when the input arrives. The notion of simultaneity is captured by the concept of event, which is a set of simultaneous occurrence of (possibly valued) signals [6]. A full definition of the language, as of version 5.91, can be found in [1].

Reactive systems are often used to control safety-critical systems. They therefore require rigorous design methods, and formal verification must be considered [6]. Constructing and verifying a formal specification of the system is very potent in detecting errors already at the formal specification development phase [9]. However, there still is a risk that there would be inconsistencies between the formal specification and the eventual product. Even while the specification is formally verified, the product itself may still be erroneous, and we want to verify that the program satisfies its safety requirements.

Verification by observers [3] is an approach to verify code. Observers are program modules monitoring the program, testing that a property is satisfied and broadcasting specific signals when the property is violated. The observers are composed in parallel to the original program, and the resulting program is compiled using an ESTEREL compiler into a finite automaton. The properties of the automaton are verified using tools such as the X ESTEREL VERIFICATION ENVIRONMENT (XEVE) [3], reducing the verification problem to reachability problem in finite automata—finding if there exists an execution trace from the initial state to a state emitting one or more of those special observer signals.

The XEVE verification environment requires the program to be compiled into Berkeley Logic Interchange Format (BLIF), a logic-level hardware hierarchical circuit description in a textual form; however, ESTEREL compiler, as of version 5.91, can either compile pure ESTEREL programs into BLIF files without changing their semantics, or, using the -soft option, abstract the data and compile only the control aspect into ESTEREL [2, Section 2.3.4]. Pure ESTEREL programs only handle pure signals, i.e., they involve no valued signals, types, constants, functions, procedures, tasks, or variables [2]. In this work we collectively refer to valued signals, sensors, and variables as valued objects. The problem with compiling a non-pure program into a BLIF circuit using the -soft option is that programs whose correctness depends on run-time values might be disqualified by XEVE as "unsafe."

Yet many control schemes, such as signal processors and closed-loop feedback controllers, receive numerical inputs, conduct numerical calculations, and emit numerical outputs. We purify such programs to allow automatic verification of their properties. This is a transformation of ESTEREL programs handling valued objects into Pure ESTEREL programs. It abstracts an unbounded, concrete system that handles data by replacing objects that take values from theoretically-infinite domains with pure signals to receive a finite system. The abstraction

\[\text{SBMF}\]
preserves the external observable behavior of the original program, i.e. there is a correspondence in terms of inputs, outputs, and timings between the two programs. Usually, abstraction-based proof techniques are sound but not complete, since the abstraction is done such that every property proven to be satisfied by the abstract system has a concrete version which holds on the concrete system, yet the other way around is unnecessarily true [4]. However, verification using our abstraction technique is both sound and complete, since the abstract program adds no new behaviors to those displayed by the concrete program; in particular, every pure signal is emitted by the observer-augmented concrete program if and only if it is emitted by its purified version. The abstraction alters the semantics of a few operations; however, these changes remain internal, i.e. the interaction with the environment is unaffected.

We suggest an algorithm that automatically purifies a certain group of programs and we characterize the class of programs verifiable using our technique. The main contribution of this work is extending the class of programs verifiable using observers with NEVE. We have implemented a prototype tool that purifies ESTEREL programs based on the algorithm.

2 ESTEREL Program Purification

In this section we describe how to transform a program that complies with certain requirements, into a pure program with an identical external observable behavior. We describe only the fundamentals of the algorithm.

We want to verify several safety properties of a given program using observers that monitor the system’s behavior and emit special signals once one of these properties is violated. The observers are assembled in threads parallel to the main program. We would be able to verify that these signals are never emitted by compiling the observer-augmented program into BLIF format; however, we cannot do so as long as the program handles data other than pure signals.

The algorithm removes from the observed program all variables, sensors, and valued signals. Instead, we represent their values and statuses using pure signals. The value of a Boolean-valued object is represented by a single pure signal (presence represents true, while absence stands for false). As for numerical-valued objects whose values range over infinite domains, we partition their domains into non-overlapping intervals, which we hereby call ranges, in such way that two goals are achieved. The first one is being able to decide any condition containing an occurrence of some numerical valued object by knowing the range within which that object’s value resides. For example, the condition \( ?s > 3.0 \) for a sensor \( s \) (\(?s\) is the current value of \( s \)) can be decided if the information whether \( ?s \in (-\infty, 3.0) \) or \( ?s \in (3.0, +\infty) \) is available. The second goal is maintaining relationships between dependent objects. For instance, for the assignment \( v := a * ?x + b \), where \( v \) is a variable, \( x \) is a valued signal and \( a \) and \( b \) are literal constants, we can determine the range within which the \( v \)'s value

\(^3\) Due to lack of space we omit some details. The full description can be found in [8].
would reside following the assignment based on the range of \( x \). E.g., for \( v := 2 \times ?x + 1 \) where \(?x\) is in (1, 3], the value of \( v \) is in \((2 \cdot 1 + 1, 2 \cdot 3 + 1) = (3, 7)\). Note that actually we calculate the ranges for \( x \) given the partition for \( u \).

The partitioning process starts from constant values assigned to the valued objects and Boolean data expressions in which they occur. It continues based on dependencies between valued objects, i.e., relations between valued objects formed when one object’s value is assigned to another, for example, the variable assignment statement \( v := 3 \times u + 1 \) creates a dependency between variables \( v \) and \( u \).

Using the calculated ranges we define pure signals that represent in the abstract system the values and statuses of valued objects from the concrete system. For example, an occurrence of an input signal \( X \) carrying a value \( x \) which resides in the \( i \)-th range calculated for \( X \) becomes an event in the abstract program in which the pure input signal \( R_{i,X} \) is received from the environment.

The operations that handle data in the concrete program are replaced with pure signal operations having identical results (up to employing pure signals where valued objects are used in the original program).

Our method supports only programs that compile and execute without errors and that comply with several constraints. These constraints are required for maintaining relationships between valued objects and for replacing numerical and Boolean calculations with pure signal operations. Some are due to technical issues with aspects of our abstraction process conflicting with the synchrony hypothesis. One must remember that one way to allow automatic methods to check a non-trivial property is to reduce the power of the language or, equivalently, reduce the class of program verifiable using the method.

Our algorithm and the tool also make several additional assumptions about the program in order to simplify the solution. These assumptions usually concern Esterel syntactic sugaring instructions. However, they do not reduce the expressive power of Esterel: each assumption can be attained by replacing the original construct by a semantically-equivalent construct, and if the program has not been initially developed complying with these assumptions, there exist automatic procedures to simplify the original program. The assumptions are listed and explained in [8].

Combining the verification power of XEVE with the transformation of non-Pure Esterel programs into Pure Esterel programs, we can verify safety properties of a larger family of programs. Various robot behaviors appearing in [7] can be implemented in Esterel, processing pure signals or requiring sufficiently simple calculations, and thus our method can be applied to them.

In [8] we characterize the set of programs to which our method is applicable and present the application of our method to various control programs. For example,
- Bang-bang controllers\(^4\) such as a temperature controller system turning on and off boilers to regulate the temperature in a chamber.
- Proportional controllers\(^5\), e.g., a program regulating the speed of a vehicle by commanding the motor to accelerate in proportion to the difference between the target speed and the current speed.

Several examples of robot behaviors, demonstrating the application of the method to mobile robot programming following the \textit{reactive control paradigm} are also provided in [8]. This paradigm, based on animal models of intelligence, decomposes the overall action of the robot by behavior, allowing handling multiple goals and multiple sensors, increasing robustness and extensibility [5]. By combining various behaviors and control algorithms, complicated control systems to which our method is applicable can be derived. The examples in [8] include original program code, abstract program code, observer code, simulations, and verification results.

3 Discussion

The program abstraction that we have implemented has an advantage over complete data abstraction performed when providing the \texttt{-soft} flag to the \textsc{Esterel} compiler, since it avoids adding behaviors not displayed by the original program. For example, consider the following code portion switching on and off an actuator based on a numerical input through a sensor \texttt{S}:

```
if \texttt{S} < 90.0 then emit \texttt{On} end if;
if \texttt{S} > 110.0 then emit \texttt{Off} end if
```

The \textsc{Esterel} compiler generates with the \texttt{-soft} flag a circuit in which both conditions can be satisfied at the same time, therefore it can emit both \texttt{On} and \texttt{Off} at the same reaction. However, no two range signals\(^6\) representing \texttt{S} can occur simultaneously in the purified program when using our abstraction, hence both conditions cannot be satisfied at the same time.

By giving up completeness, the full potential of the technique developed is realized. Applying the technique to parts of the program that fulfill the constraints

\(^4\) A bang-bang controller is a feedback controller that switches abruptly between two states [7]. The controller receives a measured quantity of interest, and outputs a certain value if that quantity is above a certain threshold, and a different value otherwise.

\(^5\) A proportional controller is a closed-loop feedback controller whose control signal is proportional to the \textit{error} – the difference between the \textit{set point}, also known as the \textit{reference} (the ideal point) and the measured quantity under control, i.e., the control signal is calculated by multiplying the \textit{error signal} by a \textit{gain} [7].

\(^6\) In the abstraction we represent a sensor \texttt{S} using pure input signals, one for each range calculated for \texttt{S}: \texttt{R1\_S}, \texttt{R2\_S}, ..., \texttt{Rn\_S}, which we call \textit{range signals}. We declare an exclusion relation among the range signals such that the environment is incapable of providing more than one range signal at a time, and add a code segment that internally-emits the first range signal in every instant in which the other are absent, such that one is always present.
while letting the ESTEREL compiler remove the rest of the data when compiling
the program into a circuit can produce a more precise over-approximation than
total control-based abstraction. This is especially useful when the system con-
sists of several sub-systems, some of which fulfilling the constraints while others
not. An example of a system comprised of a PID controller\(^7\) and a limit switch is
provided in [8]. The limit switch shuts the process down by cutting off the PID
controller’s output once an undesired limit is reached. After the measured value
drops back to the safe zone, the switch can be manually reset in order to reac-
tivate the control system. An observer helps verifying the safety of the system
by emitting a special signal whenever the program emits the output signal while
the measured value is higher than some critical threshold. Not only the calcula-
tions performed by the PID controller are not supported by our technique, but
also the program takes the set point, clock interval, and gains from constants
defined in the host language. The abstraction performed by the ESTEREL compi-
ler alone produces a circuit which XEVE reports to possibly emit the observer
signal. However, using our technique to abstract the safety limit switch and the
observer, which comply with the requirements of our techniques, and letting the
compiler abstract the rest of the program creates a circuit which never emits the
observer signal, as XEVE guarantees.

In future versions we intend to ease or completely overcome some of the con-
straints found in the current version and integrate our tool with other tools used
for observer-based verification of ESTEREL programs.

References

1. G. Berry. The Esterel v5 Language Primer, Version v5_91. Centre de Mathéma-
2. G. Berry et al., The Esterel v5_91 System Manual. Centre de Mathématiques
3. A. Bouali. Xeve, an Esterel verification environment. In CAV, volume 1427 of
Expected submission October 2014. The thesis draft and the prototype tool are
IEEE standardization proposal, 2005.

\(^7\) A Proportional-Integral-Derivative PID controller is a controller whose control signal
is the sum of three terms: one proportional to the error, one proportional to the
derivative of the error, and one proportional to the integral of the error [7].