UiO : Department of Informatics
University of Oslo

Proceedings of the PhD Symposium at iFM'18 on Formal Methods: Algorithms, Tools and Applications

(PhD-iFM'18)

Erika Ábrahám and Silvia Lizeth Tapia Tarifa (Eds.)

Research report 483, August 2018

ISBN 978-82-7368-435-6
ISSN 0806-3036

Contents

Resource-Aware Virtually Timed Ambients

Einar Broch Johnsen, Martin Steffen, Johanna Beate Stumpf, Lars Tveito

Department of Informatics, University of Oslo, Norway
{einarj,msteffen,johanbst,larstvei}@ifi.uio.no

Abstract. Virtually timed ambients is a calculus of nested virtualization, which models timing and resource consumption for hierarchically structured virtual machines. This structure may change dynamically to support load-balancing, migration, and scaling. This paper introduces resource-awareness for virtually timed ambients, which enables processes to actively query the system about the resources necessary for a task and to reconfigure accordingly. Technically we extend virtually timed ambients with context-expressions using modal logic operators, give a formal semantics for the extension, and define bisimulation for resource-aware virtually timed systems. The paper also provides a proof of concept implementation in Maude and a case study involving dynamic auto scaling. For further details, please refer to the full paper appearing in the iFM 2018 conference proceedings.

Task Planning with OMT: an Application to Production Logistics

Francesco Leofante[1], Erika Abraham[1], Armando Tacchella[2]

[1]RWTH Aachen University, Germany
[2]University of Genova,Italy
leofante@cs.rwth-aachen.de, abraham@informatik.rwth-aachen.de,
armando.tacchella@unige.it

Abstract. Task planning is a well-studied problem for which interesting applications exist in production logistics. Planning for such domains requires to take into account not only feasible plans, but also optimality targets, e.g., minimize time, costs or energy consumption. Although there exist several algorithms to compute optimal solutions with formal guarantees, heuristic approaches are typically preferred in practical applications, trading certified solutions for a reduced computational cost. Reverting this trend represents a standing challenges within the domain of task planning at large. In this paper we discuss our experience using Optimization Modulo Theories to synthesize optimal plans for multi-robot teams handling production processes within the RoboCup Logistics League. Besides presenting our results, we discuss challenges and possible directions for future development of OMT planning. For further details, please refer to "Task Planning with OMT: an Application to Production Logistics" appearing in iFM 2018

A Formal Framework to Unified Metamodel for Consistent Transformation

Jagadeeswaran Thangaraj[1,2] and Senthilkumaran Ulaganathan[2]

[1] Maynooth University, Maynooth, Ireland
[2] VIT University, Vellore, TN, India
Jagadeeswaran.Thangaraj@mu.ie

Abstract

We propose the development of a formal framework to unified metamodel within which models of class artefacts can be shared between different phases in a model-oriented environment. Model transformation approaches use different metamodels to represent source and target model of the system. This paper investigates for a unified metamodel when they share set of core representations in different phases and checks the possibilities for multidirectional transformation for code generation, upgradation and migration purposes. We envisage that the construction of an institutional framework for unified metamodel will not only increase the reusability of class artefacts but also provide a foundation for the consistent, independent and multidirectional transformation.

1 Introduction

Recently, autonomous systems development approaches get more attention. These help the developers to build software systems from requirements phase through maintenance. A model-driven development based technique is one of the approaches to software development. These play a role in both forward and reverse engineering of development part, such as, software design through implementation. In software design, developers model a system's design according to the client's requirements and they generate code using this design using some transformation techniques. In some cases, they transfer code implementation to design through reverse engineering for upgrading or verifying the correctness of the system they have developed, or transferring into other languages for further developments. The objective of this approach is to increase productivity, reduce time consumption and be cost effective.

2 Background

Object Managements Group's Model Driven Architecture (MDA) [8] relies on a set of concepts being models, modelling languages, metamodels and model transformations. A model provides a full description and overview of the system to be developed. A metamodel defines the abstract syntax of models and the interrelationships between model elements. In MDA, metamodels play a key role. Metamodeling is a technique for constructing properties of the model in a certain domain. The metamodel defines the general structure and characteristics

of real world phenomena. Therefore, a model must conform to this protocol of the metamodel similarly to how a grammar conforms for a programming language. Model transformation is a mechanism for transforming one model into another model, based on some transformation rules or programs. A transformation engine performs the transformation using corresponding program which reads one source model conforming to a source metamodel and writes a target model conforming to a target metamodel.

2.1 A Unified Metamodel

Recently, many researchers are working towards developing a unified representation to generate code from design when sharing core elements in related formalisms [3][2][9]. We have introduced a unified metamodel which has unified properties of source and target metamodels of design to implementation. In model transformation, source metamodel is different from target metamodel. Unified metamodel is based on the intersection of both source (USE [6]) and target (Spec# [7]) representations. The unified metamodel has same values for related properties for source and target metamodel as shown in Fig.1. Our approach also provides two main applications in model transformation such as: Model reusability and Multidirectional transformation.

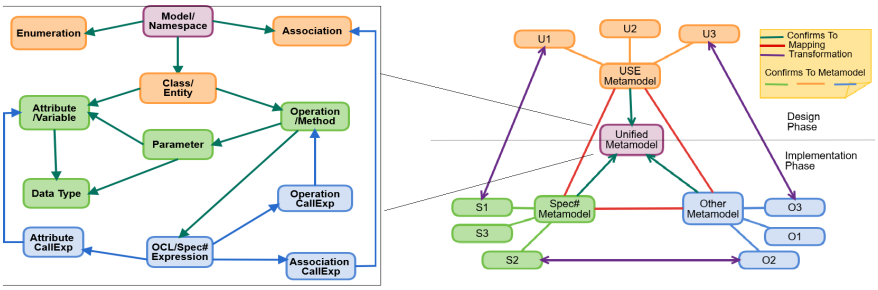


Fig. 1. Model transformation using unified metamodel

3 Our proposal: An Institution for Unified Metamodel

The question that this proposal addresses is: how to develop a formal framework to unified metamodel that allows software developers to reliably re-use models between different software representations in a model-oriented environment. The research aim of this proposal is to establish a formal framework of a unified metamodel within which these artefacts can be shared between programs and models in a model-oriented environment in order to avoid lost of constructs. Such a framework should provide a precise, formal definition of how the class artefact is constructed from unified metamodel. The framework should also be able to provide for the heterogeneity in the formalisms used, since both the software representation and the derived artefact may be based on different semantic or

models. The theory of institutions has been identified as a mechanism by which these goals can be achieved. Institutions have been defined for many logics and formalisms, including programming-related formalisms[1]. Institutions allow you to build up specifications in a formalism-independent manner using some basic constructs from category theory. Specifically, we propose to investigate this in the context of class invariants, building on our research strengths; these two artefacts are *ownership types* and *expose* blocks. These artefacts represent vital elements of the software verification process.

4 Significance

The models are defined in terms of their associated modelling languages, defined as metamodels which are themselves (typically) defined in terms of some meta metamodelling language. These metamodels also provide the basis for defining transformations between models, given a suitable transformation language. It is still easy for meaning to become lost or blurred when unification of metamodels in a model-oriented environment. One relatively simple example of a model transformation is reverse-engineering a USE class specification from a piece of Spec# code. This research focuses primarily on reasoning about ownership types [4] that are specified in terms of the concrete data and on the complications that arise due to unification. However, formally relating ownership types across multiple model representations requires a shared understanding of the model semantics. While significant advances have been made, supplying a formal framework for model-oriented development does still need more concentration in order to support consistent, independent and multidirectional transformation.

In conclusion, we have successfully defined unified metamodel for USE and Spec# and proved the required properties to reuse models and multidirectional transformation [5]. Our current task is that of implementation of institutional representation for unified metamodel. A significant future challenge is the integration of proofs for model instances generated by institutional morphisms of unified metamodel in corresponding formalisms.

References

1. D. Sanella and A. Tarlecki, “Foundations of Algebraic Specification and Formal Software Development”. Springer, 2012.
2. Fayoumi, Amjad and Kavakli, Evangelia and Loucopoulos, Pericles, “Towards a Unified Meta-Model for Goal Oriented Modelling”, In EMCIS 2015.
3. Huseyin Ergin, Eugene Syriani, “A Unified Template for Model Transformation Design Patterns”, In Patterns in Model Engineering, co-located with STAF 2015.
4. Jagadeeswaran.T, Senthilkumaran.U, “Introducing Ownership Types to Specification Design”. In IJSER DOI:https://dx.doi.org/10.14299/ijser.2017.08, 2017.
5. J.Thangaraj, S.Ulaganathan, “Towards Unified Metamodel for Multidirectional Transformation”, In Journal of Engineering and Applied Sciences, In press.
6. Martin Gogolla, Fabian Büttner, Mark Richters, “USE: A UML-based specification environment for validating UML and OCL”, Science of Computer Programming 07
7. Mike Barnett, Rustan Leino, Wolfram Schulte, “The Spec# programming system: An overview”, In CASSIS 2004: Springer, 2004.
8. OMG: Object Management Group:MDA Guide, version 2.0. ormsc/2014-06-01, http://www.omg.org/mda/ 2014.
9. S. Sepúlveda, C. Cares and C. Cachero, “Towards a unified feature metamodel: A systematic comparison of feature languages”, In CISTI 2012, Madrid, pp.1-7, 2012.

Model-based Testing of Distributed Systems and Protocols with the Coloured Petri Nets

Rui Wang

Department of Computing, Mathematics, and Physics
Western Norway University of Applied Sciences
Email: {rwa@hvl.no}

Abstract

Building cloud computing services and implementing distributed systems which require fault-tolerance and correctness involve complex distributed protocols. For such complex services and systems, ensuring availability and correctness is challenging, particularly when concurrency and communication must be handled [5]. Distributed systems and protocols can rely on a quorum system [11] to achieve fault-tolerance, however there are also challenges to implement fault-tolerance correctly. Moreover, implementing test suites for such distributed systems and protocols is also a complex and time-consuming task. *Model-based testing (MBT)* is a promising approach to software testing. It can help to improve the correctness of the design and implementation of such distributed systems and protocols. We investigate the use of Coloured Petri Nets (CPNs) for MBT of complex fault-tolerant distributed systems and protocols. Recently, our work has been done with the CPNs to apply MBT to a single-decree Paxos protocol implemented by the Go language and the Gorums framework. In this paper, we use it as an example to discuss our MBT approach with the use of CPNs for quorum-based distributed systems and protocols. Our experimental evaluation shows that we have not only obtained high code coverage for our Paxos implementation, but also found several implementation bugs by using our MBT approach.

Model-based testing and Coloured Petri Nets

Model-based testing (MBT) [10] is a powerful approach for testing software. It is based on the idea of using a model of the system under test (SUT) to generate test cases and oracles, so that a test adapter can use the generated test cases to execute the SUT and compare the results against the generated test oracles. The goal of MBT is validation and error-detection by finding observable differences between the behavior of the implementation and the intended behavior of the SUT, as defined by the testing model. As part of our current research effort, we are investigating the application of MBT on protocols for state machine replication (SMR). SMR is a core technique for developing fault-tolerant distributed systems and protocols that can tolerate a bounded number of server failures. Coloured Petri Nets (CPNs) [4] is a formal modeling language for distributed systems. It combines Petri Nets and the Standard ML [9] programming language. Petri Nets provides the foundation for modeling concurrency, communication, and resource sharing, while Standard ML gives the primitives for compact data modeling and sequential computations. Furthermore, the construction, simulation, validation, and verification of the CPN models are supported by CPN tools [1]. CPNs has a strong track record for modeling distributed systems, and enables compact modeling of data and data manipulation which is required for message modeling and concrete test case generation. Furthermore, CPNs has mature tool support for both simulation and state space exploration, which is important to implement our approach. Therefore, we use CPNs as the foundation of our MBT approach.

Recently, we have developed the MBT/CPN library [12] [8] that extends CPN Tools with support for model-based test case generation. By using it, we can further explore the application of CPNs for MBT of distributed systems and protocols.

Quorum Systems and Gorums

Replicated distributed systems and services can rely on a quorum system [11] to achieve fault-tolerance. That is, to access the replicated state, a process only needs to contact a quorum, e.g. a majority of the processes. In this way, a system can provide service despite the failure of some processes. However, communicating with and handling replies from sets of processes often complicates the protocol implementations. Gorums [7] is a library whose goal is to simplify such development effort by providing two core abstractions: a quorum call abstraction and a quorum function abstraction. The former is used to invoke a set of RPCs on a group of processes and to collect their responses. Gorums builds on the gRPC remote procedure call framework [2] developed by Google. The latter can process responses and determine if a quorum has been obtained. These abstractions help to simplify the main control flow of protocol implementations.

System Under Test and Testing Approach

We have used the Gorums framework to implement a single-decree Paxos distributed consensus protocol in Go programming language [3] as the SUT. Paxos [6] is a fault-tolerant consensus protocol that make it possible to construct a replicated service using a group of server replicas. The single-decree Paxos protocol system allows a collection of replicas to operate as a coherent group and to agree on a common value, while tolerating the failure of some of its members. The protocol system consists of proposers, acceptors, a failure detector and a leader detector. Among them, the proposer and acceptor are often known as Paxos agent roles [6] [7]. The proposers can propose values for consensus; acceptors accept a value among those proposed; then each replica can learn the chosen value. A Paxos replica may take on multiple roles: in a typical configuration, all replicas play all roles. Paxos is safe for any number of crash failures, and can make progress with up to f crash failures, given $n = 2f + 1$ acceptors. In addition, the failure detector detects the failure of replicas and the leader detector elects a new leader if the current leader failed. The Paxos protocol operates in rounds, and due to asynchrony and failures, a consensus protocol such as Paxos may need to run several rounds to solve consensus. Every round is associated with a single proposer, which is the leader for that round. Also, every round usually runs in three phases: Prepare, Accept, and Commit. The protocol starts with the Prepare phase that a proposer sends a $[Prepare]$ message to the acceptors and collects at least $f+1$ $[Promise]$ messages. After this proposer collects all $[Promise]$ messages, it becomes a leader to starts the Accept phase. During the Accept phase, the proposer (leader)

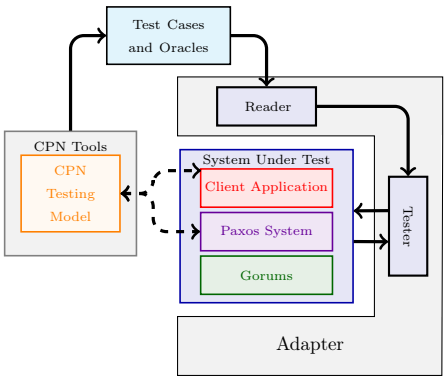


Figure 1: The SUT and testing approach.

sends [*Accept*] messages for some value v to the acceptors, who respond by sending [*Learn*] messages back in order to acknowledge the value v to the proposer. Then, the Commit phase starts by the proposer sending the decided value in [*Commit*] messages to all replicas. The communication between the different Paxos agents has been implemented based on the quorum abstractions provided by the Gorums framework. Specifically, the communication takes the form of quorum calls, one for each of the Paxos phases: Prepare, Accept, and Commit. Additionally, we have implemented a client application to access the protocol system. To test our system, we have designed a corresponding CPN testing model for such a Paxos consensus protocol system with clients. The constructed CPN testing model is comprised of 23 hierarchically organized modules. Also, we have applied MBT/CPN tool to generate test cases and oracles from the constructed CPN model. For the execution of test cases, we implemented a test adapter in Go, which consists of a reader and a tester. The reader reads test cases and oracles from XML files generated by the MBT/CPN tool; the tester uses these test cases to control the execution of the SUT and perform tests, then, it compares the test results against test oracles. Fig. 1 shows the SUT and an overview of our testing method.

Performance Evaluation and Experimental Results

To perform an evaluation of our testing approach, we consider the code coverage obtained by performing both unit tests and system tests, with a test adapter. The unit tests are used to test the central logic used to implement the single-decree Paxos protocol, such as quorum functions. The systems tests can test the complete implementation and Gorums library with the client application. In addition to testing success scenarios of the Paxos protocol system, we also test scenarios with different types of failures. This includes forcing the failure detector to timeout, triggering a new leader to be promoted. In this way, we can test leader changes and fault tolerance of the Paxos protocol. Moreover, our MBT/CPN tool can generate test cases based on either state-space exploration or simulation. However, for testing our Paxos implementation, we used simulation based test case generation due to the complexity of the CPN model. We have configured the Paxos system with 3 and 5 replicas and generated 1, 2, 5, and 10 simulation runs of the CPN model. We did not increase the number of simulation runs further, since we did not see any increase in the number of test cases from 5 to 10 simulations. For system tests, we have generated up to 38 test cases; for unit tests, we have generated up to 424 test cases. The test case generation for each configuration took less than 10 seconds, and the execution of each test case took less than one minute.

The results show that, for the unit tests, we can obtain around 90 % of the statement coverage for quorum functions, while, for the system tests, we can obtain 100 % of the statement coverage for quorum functions and 83.9 % of the statement coverage for quorum calls. Moreover, for the system tests of the Paxos core components, the Proposer module's statement coverage reaches 97.4 %, while the statement coverage of the Acceptor module is up to 100 %. The statement coverages of the Failure Detector and Leader Detector modules reach 75.0 % and 91.4 %, respectively, and the statement coverage of the Paxos replica module reaches 91.4 %. For the Gorums library as a whole, the highest statement coverage reaches 51.8 %. The results considered above validate that the implementation of the single-decree Paxos system and the Gorums framework works in both correct scenarios and scenarios involving failures of replicas. We have noticed that the coverage results of the Gorums library is relatively lower. That is because Gorums library contains code, such as various auxiliary functions and error handling code, that are not used by our current implementation.

Furthermore, we have discovered bugs in the implementation of the Paxos protocol. For

example, the leader detector elects a wrong leader; only the leader's failure detector is executed; clients cannot receive responses from the Paxos replicas; the Paxos system can only handle one request from one client. These bugs are not captured by manually written table-driven tests in Go. This shows that how our MBT approach can detect non-trivial programming errors in complex distributed systems protocols. For our future work, we plan to extend our current Paxos implementation to multi-decree Paxos and perform MBT for such complex system.

References

[1] CPN Tools. CPN Tools homepage. http://www.cpntools.org.

[2] Google Inc. gRPC Remote Procedure Calls. http://www.grpc.io.

[3] Google Inc. The Go Programming Language. https://golang.org.

[4] K. Jensen and L. Kristensen. Coloured Petri Nets: A Graphical Language for Modelling and Validation of Concurrent Systems. *Communications of the ACM*, 58(6):61–70, 2015.

[5] Jepsen. Distributed Systems Safety Analysis. http://jepsen.io.

[6] L. Lamport. The part-time parliment. *ACM Trans. on Comp. Syst.*, 16(2):133–169, 1998.

[7] T. E. Lea, L. Jehl, and H. Meling. Towards New Abstractions for Implementing Quorum-based Systems. In *37th IEEE Intl. Conf. on Distributed Computing Systems (ICDCS)*, 2017.

[8] MBT/CPN repository. https://github.com/selabhvl/mbtcpn (Jan 2018).

[9] Standard ML. Standard ML Family GitHub Project. http://sml-family.org.

[10] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability*, 22:297–312, 2012.

[11] M. Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Morgan and Claypool, 2012.

[12] R. Wang, L. Kristensen, and V. Stolz. MBT/CPN: A Tool for Model-Based Software Testing of Distributed Systems Protocols using Coloured Petri Nets (accepted). In *12th International Conference on Verification and Evaluation of Computer and Communication Systems (VECoS 2018)*, Grenoble, France, 2018.

Non-intrusive MC/DC Measurement based on Traces*

Faustin Ahishakiye[1], Felix D. Lange[2]

[1] Western Norway University of Applied Sciences, Bergen, Norway Email: *fahi@hvl.no*
[2] Institute for Software Engineering and Programming Languages, Universität zu Lübeck, Lübeck, Germany Email: *lange@isp.uni-luebeck.de*

1 Introduction and Motivation

To produce fail-safe and low risk software for safety critical systems, certification standards, for example the DO-178C [7] in the domain of avionic software systems, are used by certification authorities, like the Federal Aviation Administration (FAA) and the European Aviation Safety Agency (EASA), to approve and ensure that the software used follows certain software engineering standards. DO-178C requires that structural coverage analysis is performed during the verification process mainly as a completion criterion for the testing effort. Coverage refers to the degree to which the source code of a program is executed when a given test suite runs. Software level C (major effect) requires *statement coverage*, software level B (hazardous effect) requires *decision coverage* and software level A (catastrophic effect) requires *modified condition/decision coverage* (MC/DC) [8]. While logical errors and errors based on wrong control flow structures are hard to detect with weaker coverage criteria such as statement coverage, strong coverage criteria like multiple condition coverage (MCC), that require every possible combination of all conditions in a decision, lead to an exponential growth of test cases. Therefore, MC/DC is a recommended candidate and subsumes the existing coverage criteria because it is sensitive to the complexity of the structure of each decision [3] and requires less test cases, compared to exhaustive MCC. MC/DC is defined as follow [8]:

"Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has shown to independently affect that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome".

Usually MC/DC is measured by instrumenting the source code in order to observe information about taken paths, executed statements and evaluated conditions. Instrumentation is intrusive (causes disruption) and needs to be removed after the coverage measurement while it has to be proven that the behavior of the code with and without instrumentation does not change. Alternatively it is possible to leave the instrumentation in the released code but that would consume resources that are valuable in embedded systems widely used in this domain.

In this paper, we describe a novel approach of how MC/DC can be measured non-intrusively by analyzing program traces obtained from modern hardware-based tracing facilities such as IntelPT). Our approach is based on the idea that every condition in the source code is translated to a conditional jump on the object code level [2]. Program traces contain information about taken jumps during the execution and make it possible to reconstruct the evaluation of each condition without instrumentation.

*This work was partially supported by European Union's Horizon 2020, through project 732016, COEMS (Continuous Observation of Embedded Multicore Systems).

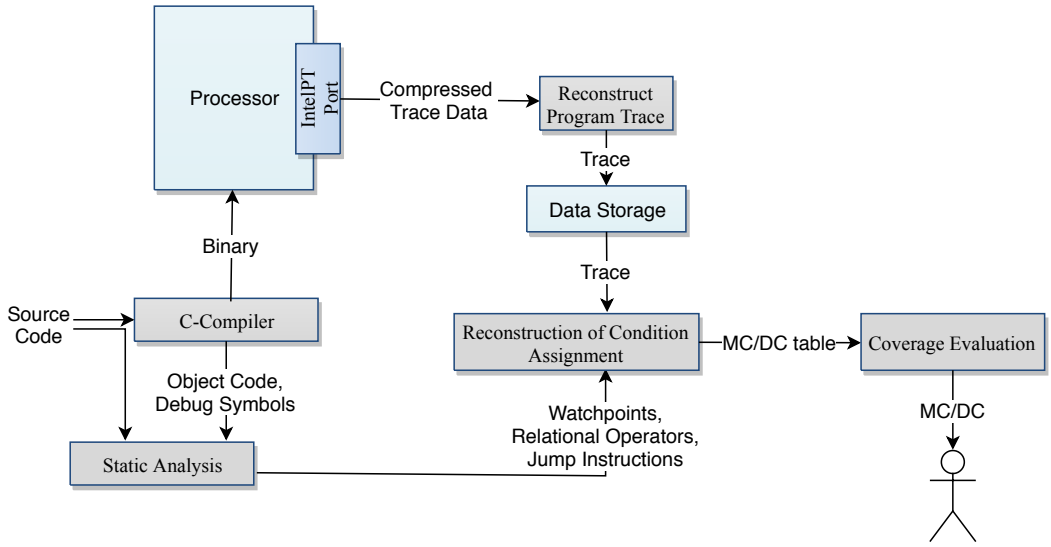


Figure 1: Overview of the implementation.

2 MC/DC Tool Implementation

Our tooling [6] consists of the following parts: a frontend that detects so-called decision (boolean expressions) that are used in conditionals in C source code, a mapping from conditional jumps in the object code back to those decisions, and an analysis that computes satisfaction of the MC/DC coverage relation on those decisions from an execution trace as shown in Figure 1. This analysis takes as input a stream of instruction addresses decoded from IntelPT-data recorded while running the software under test.

In order to find decisions in the source code we use the *Abstract Syntax Tree* (AST) representation provided by LLVM. With *LibTooling* and the *AST-matcher* [9], our tool detects all conditions in if-, for- and while-statements in the source code and gathers corresponding information as line and column numbers and then-statements.

The direct mapping between conditions and conditional jumps in the object code is possible by utilizing debug symbols provided by the compiler. We use *clang 5.0* or better since this compiler provides rich debug symbols containing even line and column information (compiler option `-g -XClang -dwarf-column-info`). Combined with the detected decisions, we then can detect all conditional jumps that are useful for measuring MC/DC based on traces. The mapping has been implemented with Python and is usable in a command line or via a Graphical User Interface (GUI).

Nowadays, different program tracing technologies are available including: 1) ARM CoreSight used by ARM processors for debug and trace for multicore SoCs [1]; 2) Intel Processor Trace (IntelPT), an extension of Intel Architecture that traces program execution (every branch) with low overhead [5]. We use IntelPT to generate a trace of the execution of a program. The technology is widely available, which makes it suitable for this proof-of-concept tool. With *perf*, the Linux-kernel provides an easy-to-use implementation of the recording and reconstruction of IntelPT-traces. Because the reconstructed trace becomes excessively large even for short executions time, we filter the trace and only store those instructions that contain information that is relevant to measuring MC/DC.

3 Performance Evaluation and MC/DC Measurement

The performance evaluation of our MC/DC tool is based on C programs containing conditionals with Boolean expressions. The tool chain of detecting all decisions, mapping conditions to conditional jumps, running and tracing the program and measuring MC/DC based on the trace is used via a GUI.

We show the detected decisions and visualize the measured coverage in the source code, calculate which decisions have been covered according to the MC/DC criterion. This can help developers in devising additional tests. For a decision with n conditions and which has been evaluated m times, MC/DC is measured by filling the table with m rows and $n + 1$ columns with the reconstructed evaluations of the conditions (the last column shows the outcome of the decision) and comparing pairwise all the rows with different outcomes. If two compared rows contain a different entry (false and true) for exactly one condition, these two assignments show the independent effect for this condition. More details about MC/DC tool can be found on the following link: https://www.coems.eu/non-intrusive-mc-dc-measurement-ifm/.

Our tool shows the feasibility of measuring MC/DC without instrumentation based on program traces. However, there are some limitations. It is necessary to disable optimizations during the compilation because even on the first optimization level many conditions are not directly translated into conditional jumps but into conditional moves, jump tables or indirect branches due to performance reasons. Because the program trace contains no information on how these instructions are evaluated, they cannot be used to reconstruct the evaluation of conditions. Another problem of our approach is that the trace data becomes excessively large for longer executions. To address this issue, we want to use FPGA-based hardware to record the trace [4] without using *perf*. Moreover, we will examine to what extent IntelPT can be used in terms of time and size with or without *perf*. With the experience gained from measuring MC/DC coverage, we plan to analyse coverage for concurrent programs. We also plan to use the trace information to identify hot-spots and contention in programs.

References

[1] ARM Limited. *ARM IHI 0029B: CoreSightTM Architecture Specification v2.0, issue D*, 2013.

[2] CAST. Structural coverage of object code, position paper 17. Technical report, Certification Authorities Software Team, 2003.

[3] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[4] N. Decker, B. Dreyer, P. Gottschling, C. Hochberger, A. Lange, M. Leucker, T. Scheffel, S. Wegener, and A. Weiss. Online analysis of debug trace data for embedded systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 851–856, March 2018.

[5] A. Kleen. Cheat sheet for Intel Processor Trace with Linux perf and gdb, April 2017. Available at http://halobates.de/blog/p/410.

[6] F. D. Lange. Modified Condition/Decision Coverage based on jumps, 2018. Master's thesis, available at http://www.isp.uni-luebeck.de/thesis/modified-conditiondecision-coverage-based-jumps.

[7] F. Pothon. DO-178C/ED-12C versus DO-178B/ED-12B: Changes and Improvements. Technical report, AdaCore, 2012. Available at https://www.adacore.com/books/do-178c-vs-do-178b.

[8] L. Rierson. *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. CRC Press, 2013.

[9] The Clang Team. Matching the Clang AST. Clang documentation. Available at https://clang.llvm.org/docs/LibASTMatchers.html.

A Compositional Abstraction Technique for Automated Verification of Concurrent Programs

Fauzia Ehsan[1] and Marieke Huisman[1]

University of Twente, the Netherlands

Keywords: program verification, compositional reasoning, Fastlane algorithm

1 Introduction

Modern increasingly parallel and distributed computing systems highly rely on the concepts of concurrency, distribution and modularity. Efficiency as well as reliability is a very challenging problem for these complex and dynamic software systems used for computationally intensive applications. Combining an abstraction theory with *deductive formal verification* provides a technique to reason about concurrent and non-deterministic software programs at a high abstraction level. It verifies functional correctness of a program along with data-race freedom and memory safety [1].

In order to reason about parallel programs, in earlier work, we proposed a practical abstraction theory, using the notion of process algebra and static program verification techniques [1,2]. The abstraction theory allows one to reason about an abstract model of the program instead of over a concrete model and this makes modular verification easy and manageable. Compositional reasoning analyzes the program efficiently by enabling local reasoning of a program and verifies its global behaviour [3]. We can optimize the program verification techniques and make them modular at the process algebra level by employing the concept of compositional reasoning. The aim of our work is to formalize this existing abstraction theory and to extend it with a general compositional abstraction theory for the models of concurrent and distributed software programs.

2 Approach

Our approach uses a deductive formal verification technique to specify the program behavior. *Permission based separation logic (PBSL)*, an extension of *formal Hoare Logic,* is used to annotate the program. Process algebra, a well-known technique to represent the abstract behaviour of parallel program, is used by defining actions with certain contracts to describe the atomic behaviour and these actions are combined into processes using sequential and parallel composition operators. The abstraction connects program code with atomic concurrent actions and in this way the behaviour of reactive and non-terminating programs is specified. Program behaviour is verified using automated tool support. We

plan to develop a compositional reasoning approach to reason about the abstract models, based on assume-guarantee [4], in order to verify properties over concurrent and distributed programs. We already have established a collection of sound proof rules that can be used to prove the correspondence between the concrete code and the abstract model. The proof rules are incorporated in the VerCors tool set, in order to have automated support to reason about abstractions [5].

In order to fully explore and understand our approach, the presented technique will be used to reason about an implementation of software transactional memory algorithms. We will specifically apply it to the FastLane mode of the FastLane algorithm [6,7] in which the synchronization of parallel threads on shared memory is very challenging. As a first step, we will formally specify and verify the behaviour of FastLane mode using our proposed abstraction technique in the VerCors tool set. This case study will be further discussed during the presentation. Thus, we aim at providing a practical verification technique for compositional verification of functional properties of concurrent programs, by means of abstractions.

3 Future work

As a further step, we will explore the verification of lock free data structures using program verification and abstraction. We also will investigate if we can further extend abstraction-refinement techniques for the verification of concurrent programs.

References

1. Wytse Oortwijn, Stefan Blom, Dilian Gurov, Marieke Huisman, and M. Zaharieva. *An Abstraction Technique for Describing Concurrent Program Behaviour*, pages 191–209. Lecture Notes in Computer Science. Springer, 2017.
2. M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying Functional Behaviour of Concurrent Programs.* PhD thesis, University of Twente, 2015.
3. Dimitra Giannakopoulou, Kedar S. Namjoshi, and Corina S. Păsăreanu. *Compositional Reasoning*, pages 345–383. Springer International Publishing, Cham, 2018.
4. Mihaela Gheorghiu Bobaru, Corina S. Păsăreanu, and Dimitra Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 135–148, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
5. Afshin Amighi, Stefan Blom, and Marieke Huisman. Vercors: A layered approach to practical verification of concurrent software. In *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*, pages 495–503. IEEE, 2016.
6. Gerhard Schellhorn, Monika Wedel, Oleg Travkin, Jürgen König, and Heike Wehrheim. Fastlane is opaque–a case study in mechanized proofs of opacity. In *International Conference on Software Engineering and Formal Methods*, pages 105–120. Springer, 2018.

7. Jons-Tobias Wamhoff, Christof Fetzer, Pascal Felber, Etienne Rivière, and Gilles Muller. Fastlane: Improving performance of software transactional memory for low thread counts. *SIGPLAN Not.*, 48(8):113–122, February 2013.

Stateful Behavioral Types for Active Objects

Eduard Kamburjan and Tzu-Chun Chen

Technical University of Darmstadt, Germany
kamburjan@cs.tu-darmstadt.de, tc.chen@dsp.tu-darmstadt.de

Abstract. It is notoriously hard to correctly implement a multiparty protocol which involves asynchronous/concurrent interactions and the constraints on states of multiple participants. To assist developers in implementing such protocols, we propose a novel specification language to specify interactions within multiple object-oriented actors and the side-effects on heap memory of those actors; a behavioral-type-based analysis is presented for type checking. Our specification language formalizes a protocol as a global type, which describes the procedure of asynchronous method calls, the usage of futures, and the heap side-effects with a first-order logic. To characterize runs of instances of types, we give a model-theoretic semantics for types and translate them into logical constraints over traces. We prove protocol adherence: If a program is well-typed w.r.t. a protocol, then every trace of the program adheres to the protocol, i.e., every trace is a model for the formula of its type. The full paper for this work is part of the proceedings of iFM'18.

The HyDRA Tool – A Playground for the Development of Hybrid Systems Reachability Analysis Methods

Stefan Schupp and Erika Abraham

RWTH Aachen University, Aachen, Germany
{stefan.schupp, abraham}@cs.rwth-aachen.de

Hybrid systems are systems with mixed discrete-continuous behavior. *Hybrid automata* are a popular formalism to model hybrid systems. When modelled as a hybrid automaton, the safety of a hybrid system can be analyzed by computing the set of all *reachable* states of the hybrid automaton model and intersecting it with a user-defined set of unsafe states.

Though this *reachability problem* is in general undecidable, several incomplete approaches have been proposed. One class of these approaches is based on the computation of *flowpipes* to *over-approximate* the set of all reachable states. These analysis techniques need a *representation* of state sets and operations on these representations to implement set operations like intersection, union, linear transformation etc. in an over-approximative manner. To support the fast development of such methods, we developed a `C++` library HyPro [SÁBMK17], which provides different datatypes for the representation of state sets (e.g. boxes, convex polyhedra, support functions, zonotopes), data structures for hybrid automata, and utility functions for conversion between representations, plotting and parsing.

In this abstract we report our tool HyDRA for flowpipe-construction-based reachability analysis of hybrid systems. Initially being used as a prototype to show general applicability of the library, the HyDRA tool has been continuously developed and extended with new approaches. In its current version, it provides a modularized basic reachability analysis algorithm based on flowpipe construction, which has been enhanced by several features:

Partial path refinement. In our work [SÁ18a] we developed an extension of the basic reachability algorithm which allows *counterexample-guided abstraction refinement*: using a state set representation that provides fast computations at the cost of large over-approximation error we might succeed to prove the safety along some paths but fail on some others. In the latter case we refine computations along the counterexample paths by using more precise configurations or other state set representations to resolve spurious counterexamples during analysis. The method of refinement for counterexample paths is chosen from a pre-defined ordered sequence called the *refinement strategy*. During the refinement, our approach exploits information collected during previous computations to speed up the refinement of certain paths in the induced search tree. At the moment we collect time intervals in which discrete transitions were enabled to reduce the effort spent to detect enabled transitions in following runs.

Parallelization. A recent extension of our tool described in [SÁ18b] allows us to analyze different execution paths in *parallel* to speed up the overall analysis. Our approach incorporates several techniques to reduce *synchronization* effort as well as to balance the *work distribution* among worker threads during the analysis.

Subspace decomposition. In [SNÁ17] we proposed an approach to effectively reduce the dimension of the state space in which the computations take place. The basic idea is to decompose the variable set into subsets such that all atomic constraints in the model use variables from

exactly one of the subsets; this property is called *syntactic independence*. We have implemented this approach in our tool to automatically determine syntactically independent subspaces and further select suitable analysis methods indivudually based on the dynamics of each particular subspace. In this context we have extended our portfolio of analysis methods by an approach dedicated for the efficient reachability analysis of timed automata.

Future work. Following our work on partial path refinement, we plan to develop an extension in which potential counterexamples that cannot be declared as being spurious during analysis are verified using simulation approaches [NÁCC14]. Our plan is to simulate a fixed set of traces of the given system to be able to provide more detailed information as to why a given system might be unsafe.

References

[NÁCC14] Johanna Nellen, Erika Ábrahám, Xin Chen, and Pieter Collins. Counterexample generation for hybrid automata. In *Proc. of FTSCS'13*, pages 88–106. Springer, 2014.

[SÁ18a] Stefan Schupp and Erika Ábrahám. Efficient dynamic error reduction for hybrid systems reachability analysis. In *Proc. of TACAS'18*, volume 10806 of *LNCS*, pages 287–302. Springer, 2018.

[SÁ18b] Stefan Schupp and Erika Ábrahám. Spread the work: Multi-threaded safety analysis for hybrid systems. In *Proc. of SEFM'18*, volume 10886 of *LNCS*, pages 89–104. Springer, 2018.

[SÁBMK17] Stefan Schupp, Erika Ábrahám, Ibtissem Ben Makhlouf, and Stefan Kowalewski. HyPro: A C++ library for state set representations for hybrid systems reachability analysis. In *Proc. of NFM'17*, volume 10227 of *LNCS*, pages 288–294. Springer, 2017.

[SNÁ17] Stefan Schupp, Johanna Nellen, and Erika Ábrahám. Divide and conquer: Variable set separation in hybrid systems reachability analysis. In *Proc. of QAPL'17*, volume 250 of *EPTCS*, pages 1–14. Open Publishing Association, 2017.

Machine Learning and AI Techniques for Automated Tool Selection for Formal Methods

Jannik Dunkelau

Heinrich-Heine-University, Düsseldorf, Germany
jannik.dunkelau@hhu.de

Abstract

The typical formal method workflow is aided by a broad variety of tools, each featuring a multitude of configuration options. However, selecting the right tool with suitable configuration depending on a given problem instance is a non-trivial task. Therefore my research proposal is to explore whether the process can be automated using AI techniques. Beginning with a backend selection for the model checker ProB the aim is to work towards fully automated configuration.

1 Introduction

The typical formal method workflow consists of writing specifications and analysing them using proof techniques and model checking. For both tasks, a variety of tools exists, each featuring a multitude of configuration options. However, selecting the right tool for a job and choosing the optimal configuration is a non-trivial task, even for an expert. This is in alignment of the *No Free Lunch* theorem [18, 17], and also affirmed by empirical evaluation on verification tasks as shown in [7] for the B method.

In related work, machine learning powered heuristics for an algorithm portfolio already proved to perform well for SAT instances, as shown by SATZilla [19]. For first order logic with equality, an automated heuristic selection employing machine learning [2] was developed for the E Theorem Solver [15]. As those approaches already provided promissing results in their own subdomains, I focus my research on the question whether a tool selection can be automated for more complex problems encountered in formal methods.

Section 2 outlines my research proposal, whereas my previous work on the subject is summarised briefly in Section 3.

2 Research Goals

My research proposal is to explore the applicability of artificial intelligence (AI) techniques, primarily that of machine learning algorithms, for assembling a heuristic to enhance both tool selection and tool configuration for formal verification tasks like constraint solving.

Applications for formal methods might include selection of a suitable tool for a given problem instance, attribute classification aiding in selection of different settings for model checking, or regression of estimated time needed for model checking. This poses the need for a set of suitable features on which machine learning techniques can operate. Compiling such a feature set requires extensive domain knowledge and expertise and acquiring it is a cumbersome and extensive task. However, once a satisfactory performance of a learning algorithm is achieved, the relative importance of individual features for the prediction step could further be analysed, eventually leading to a deeper insight of why specific tools perform better on given instances than others. Thus, this research could increase our understanding of the problem domain in the long run.

3 Previous Work

In previous work [5], I explored the applicability of using deep neural networks [14, 6] for achieving a selection heuristic which predicts for a given constraint the most suitable solver. Hereby I focused on the B method [1]. The problem instances were B predicates, and the selection step was to decide which backend of ProB to use.

The B method [1] consists of a formal verification language for specifying software systems which can be proven mathematically. B utilises sets for data modelling, general substitution for state modifications, and refinement calculus. ProB [11, 10, 9] is an animator, model checker, and constraint solver. At its core, ProB is written using SICStus Prolog's CLP(FD) library [3] to solve constraints posed by a given specification. Besides its native constraint solving backend, ProB offers the ability to substitute and support it with alternative backends, such as the SAT-based Kodkod Constraint Solver [16] (connected to ProB via [13]) or the SMT-based Z3 Theorem Prover [4] (connected to ProB as outlined in [8]).

I generated training sets for each experiment, having an equal representation of favoured backends and consisting of 50,000–60,000 predicates (depending on the experiment in question). Those predicates were extracted and generated from 3638 B machines stemming from different application areas and thus varying in size and complexity. The experiments consisted of

1. Classifying whether a predicate is decidable by ProB alone or not,
2. Selecting between ProB and Kodkod, and
3. Selecting between ProB and Z3.

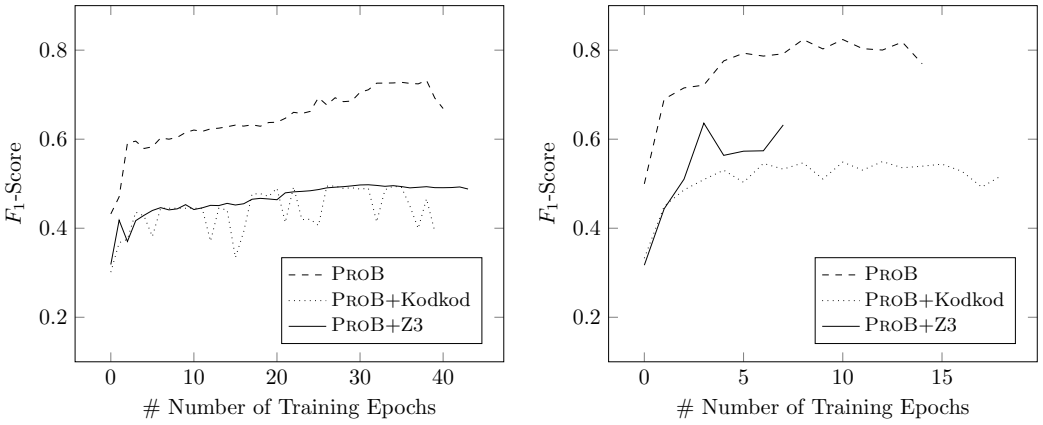
The training was conducted over three handcrafted feature sets growing in number of predicate features extracted. Those features were directly calculated over the AST of the respective B predicate and integrated domain-knowledge (by experience) of the B method.

Additionally, predicates were transformed into images to train a convolutional neural network as proposed by [12]. This approach maps characters to gray scale pixels, and fits them into a square image which is then scaled to a fixed size. As for these images no prior domain-knowledge was necessary, they served as a reference implementation to measure the relative performance of the handcrafted features.

Figure 1 presents the best trained models for each of the aforementioned experiments, conducted over the handcrafted features and the generated images respectively. The F_1-Score hereby serves as measure of performance, being the average of a model's precision and recall. Precision is the probability of a predicted backend to be the correct choice for a predicate, whereas the recall of a certain backend describes the probability that it will be predicted for predicates it is most suitable for.

It appears that those experiments between ProB and Z3 produced more robust (in terms of steady learning curves) or better performing (in terms of correctness of predictions) models than the corresponding Kodkod experiments. This might be due to the distinct algorithms employed by ProB and Z3. Conducting further experiments over these two backends might yield well-performing results.

Much more interesting however is the fact that the handcrafted features in each experiment performed worse than the corresponding image based approach. This puts further emphasis on how complex the problem domain is in the first place and how powerful of a tool machine learning is.



(a) Best models of feature based approach

(b) Best models of convolutional approach

Figure 1: Comparison of best models for each experimental question for handcrafted features and for images respectively.

4 Conclusions

The evaluation of the results and insights gathered from the aforementioned previous work lead me to believe that the approach of using AI techniques to automate a backend selection for ProB is promising. We apparently do not understand the problem domain enough to outsmart simple images of the predicates. Making use of machine learning algorithms and then extracting their gathered knowledge might lead to a deeper insight of why certain predicates are easier to decide with specific backends, allowing to further increase performance of solvers on specific problem instances. Neural networks however might not be the most promising way to achieve such knowledge, but other algorithms (e.g. decision trees and random forests) might.

On the other hand, the predicate to image translation appears to be a sensible approach for future work. Better performing models may be trained without the need to further analyse the problem domain beforehand. Using a sequence oriented model, such as that of a recurrent neural network, might be an even more sensible choice. B predicates are essentially sequences of characters, rendering such a technique an intuitive approach.

Overall, the combination of formal methods and machine learning algorithms has a lot of potential and I would like to investigate further which algorithms are applicable for different problem statements.

References

[1] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[2] James P Bridge. Machine learning and automated theorem proving. Technical report, University of Cambridge, Computer Laboratory, 2010.

[3] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In *Proceedings PLILP*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.

[4] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[5] Jannik Dunkelau. Automatic Selection of Solvers Using Deep Learning. Master's thesis, Heinrich-Heine-University Düsseldorf, Germany, 2017.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[7] Sebastian Krings, Jens Bendisposto, and Michael Leuschel. From Failure to Proof: The ProB Disprover for B and Event-B. In *Proceedings SEFM*, volume 9276 of *LNCS*. Springer, 2015.

[8] Sebastian Krings and Michael Leuschel. SMT Solvers for Validation of B and Event-B models. In *International Conference on Integrated Formal Methods*, pages 361–375. Springer, 2016.

[9] Michael Leuschel, Jens Bendisposto, Ivaylo Dobrikov, Sebastian Krings, and Daniel Plagge. From Animation to Data Validation: The ProB Constraint Solver 10 Years On. In Jean-Louis Boulanger, editor, *Formal Methods Applied to Complex Systems: Implementation of the B Method*, chapter 14, pages 427–446. Wiley ISTE, Hoboken, NJ, 2014.

[10] Michael Leuschel and Michael Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[11] Michael Leuschel, Michael Butler, et al. ProB: A model checker for B. In *FME*, volume 2805, pages 855–874. Springer, 2003.

[12] Andrea Loreggia, Yuri Malitsky, Horst Samulowitz, and Vijay A Saraswat. Deep Learning for Algorithm Portfolios. In *AAAI*, pages 1280–1286, 2016.

[13] Daniel Plagge and Michael Leuschel. Validating B, Z and TLA$^+$ using ProB and Kodkod. In Dimitra Giannakopoulou and Dominique Méry, editors, *Proceedings FM*, LNCS, pages 372–386. Springer, 2012.

[14] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, CORNELL AERONAUTICAL LAB INC BUFFALO NY, 1961.

[15] Stephan Schulz. E–a brainiac theorem prover. *Ai Communications*, 15(2, 3):111–126, 2002.

[16] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[17] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[18] David H Wolpert, William G Macready, et al. No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

[19] Lin Xu, Frank Hutter, Jonathan Shen, Holger H Hoos, and Kevin Leyton-Brown. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, pages 57–58, 2012.