A Framework for Compositional Design and Analysis of Systems

Arindam Chakrabarti

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-174
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-174.html

December 20, 2007

Acknowledgement

This Technical Report comprises my Doctoral Dissertation and is based on joint work with Prof Luca de Alfaro, Prof Dirk Beyer, Dr Patrice Godefroid, Prof Thomas A. Henzinger, Prof Orna Kupferman, Prof Rupak Majumdar, Dr Freddy Y. C. Mang, Prof Sanjit A. Seshia, and Prof Mari\"elle Stoelinga. I am also deeply grateful to Prof Marcin Jurdzi{\'n}ski and Dr Nir Piterman for many valuable suggestions, comments, insights, and enlightening discussions. I am extremely grateful to my advisor Prof Tom Henzinger, Prof Edward Lee, Prof George Necula, and Prof Jack Silver for extremely useful feedback, advice, comments, and suggestions that made this Dissertation possible.

A Framework for Compositional Design and Analysis of Systems

by

Arindam Chakrabarti

B. Tech. (Indian Institute of Technology Kharagpur) 2001
M.S. (University of California at Berkeley) 2005

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Thomas A. Henzinger, Co-Chair
Professor George C. Necula, Co-Chair
Professor Edward A. Lee
Professor Jack H. Silver

Fall 2007

The dissertation of Arindam Chakrabarti is approved.

Co-Chair	Date
Co-Chair	Date
	Date
	Date

University of California, Berkeley

Fall 2007

A Framework for Compositional Design and Analysis of Systems

by

Arindam Chakrabarti

Abstract

A Framework for Compositional Design and Analysis of Systems

by

Arindam Chakrabarti

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Thomas A. Henzinger, Co-Chair

Professor George C. Necula, Co-Chair

Complex system design today calls for compositional design and implementation. However each component is designed with certain assumptions about the environment it is meant to operate in, and delivering certain guarantees if those assumptions are satisfied; numerous inter-component interaction errors are introduced in the manual and error-prone integration process as there is little support in design environments for machine-readably representing these assumptions and guarantees and automatically checking consistency during integration.

Based on Interface Automata [54] we propose a framework for compositional design and analysis of systems: a set of domain-specific automata-theoretic type systems for compositional system specification and analysis by behavioral specification of open systems. We focus on three different domains: component-based hardware systems communicating on bidirectional wires. concurrent distributed recursive message-passing software systems, and embedded software system components operating in

resource-constrained environments. For these domains we present approaches to formally represent the assumptions and conditional guarantees between interacting open system components. Composition of such components produces new components with the appropriate assumptions and guarantees. We check satisfaction of temporal logic specifications by such components, and the substitutability of one component with another in an arbitrary context. Using this framework one can analyze large systems incrementally without needing extensive summary information to close the system at each stage. Furthermore, we focus only on the inter-component interaction behavior without dealing with the full implementation details of each component. Many of the merits of automata-theoretic model-checking are combined with the compositionality afforded by type-system based techniques. We also present an integer-based extension of the conventional boolean verification framework motivated by our interface formalism for embedded software components.

Our algorithms for checking the behavioral compatibility of component interfaces are available in our tool Chic [1], which can be used as a plug-in for the Java IDE JBuilder [2] and the heterogenous modeling and design environment Ptolemy II [3].

Finally, we address the complementary problem of partitioning a large system into meaningful coherent components by analyzing the interaction patterns between its basic elements. We demonstrate the usefulness of our partitioning approach by evaluating its efficacy in improving unit-test branch coverage for a large software system implemented in C.

Professor Thomas A. Henzinger
Dissertation Committee Co-Chair

Professor George C. Necula
Dissertation Committee Co-Chair

To my dear father, Arya Kumar Chakrabarti, and my dear mother, Minati Chakrabarti, with love and respect, as a token of gratitude for the innumerable sacrifices they have always silently made for me throughout my life.

Contents

List of Figures

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Tom Henzinger, for the incredibly immense amount of support, help, encouragement, and motivation I have always received from him. I have been immensely lucky to have had the opportunity to interact with him closely for the last six years and learn from him all that I know about formal verification. It has been my immensely good fortune to have had the opportunity to be in the wonderful research and learning environment in his research group at Berkeley.

I shall always vividly remember the first meeting I had with him on Monday 6 August 2001 when I first arrived at Berkeley, when I was wide-eyed and overwhelmed at having just met him and some of the other world-famous computer scientists among his collaborators (like Prof Orna Kupferman, Prof George Necula, Prof Alex Aiken, Prof Luca de Alfaro, among others), some of whose famous papers I had read as an undergraduate student in India, and he put me at ease in his characteristic encouraging and motivating manner. Since that day I first had the honor of meeting him, he has been a constant inspiration and a perennial source of encouragement and motivation, and has always been patient and available for discussions, even late into the night, even when he was tired after a long day, or jet-lagged after a long flight, and in spite of the tremendous constant time pressure that accompanies his position as one of the foremost Computer Scientists on earth. He has been my role model since I first interacted with him, and I have always fervently wished that a little bit of some of his brilliant qualities might rub off on me. It has been my indescribably good fortune to have gotten the opportunity to work with and learn from him, and I hope someday I will be able to inculcate in myself a little fraction of some of his brilliant qualities.

I am also extremely grateful for the constant financial support he gave me throughout my graduate studies: financial support for me; support to cover my registration fees, non-resident tuition, and other University fees; support for me to attend numerous conferences from which I gained so much exposure, experience, and breadth; and after he left Berkeley to join EPFL Lausanne in Switzerland, support for me to visit him multiple times in Lausanne to work with him. Due to his kind support I never had to worry about financial support at any point of my graduate studies and was able to focus all my time and energy on research.

My first experience of the truly fascinating intellectual stimulation provided in some of the classrooms at Berkeley occurred shortly after I arrived at Berkeley, when I took CS 263, the programming languages theory class taught by Prof George Necula. I had taken an advanced course on Theory of Programming Languages as an undergraduate student at IIT Kharagpur the previous year. However the way Prof Necula elucidated the core concepts of the subject made me look at everything in a new light and with an incomparably deeper understanding. I had a similarly fascinating intellectual experience when I attended the lectures of Prof Alex Aiken and Prof Henzinger respectively on implementation of programming languages, and on formal verification, and algorithms. It was an immensely good fortune for me to take these fabulous courses through which I discovered all that I know about program analysis, theorem proving, type systems, and model checking, and the connections between them.

I would also like to thank Prof Alex Aiken, Prof Ras Bodik, Prof George Necula, Prof Koushik Sen, and Prof Sanjit Seshia, for having created and run, along with Prof Henzinger, a very inspiring research environment in the Open Source Quality (OSQ)

research group that I had the opportunity to be a part of. Attending the weekly OSQ research meetings was an extremely valuable experience.

This dissertation is based on joint work with Prof Luca de Alfaro, Prof Dirk Beyer, Dr Patrice Godefroid, Prof Thomas A. Henzinger, Prof Orna Kupferman, Prof Rupak Majumdar, Dr Freddy Y. C. Mang, Prof Sanjit A. Seshia, and Prof Mariëlle Stoelinga. It has been a great privilege for me to have gotten the opportunity to work with and learn from such incredibly creative researchers and encouraging and inspiring mentors. I would like to thank them for for being patient, available, encouraging, and always helpful. I learnt a great deal from each of them and benefited beyond measure in the course of working on research projects with them. There are no words I have that can sufficiently express my gratitude for all the help I got from them and all I learnt from them, but Thank you! I also benefited greatly from working with and learning from Prof Marcin Jurdziński on a related research project [40] when I arrived at Berkeley. I am also deeply grateful to Dr Nir Piterman for many valuable suggestions, comments, insights, and enlightening discussions. Since I first met him in December 2000 in Delhi, India at FSTTCS 2000, and later at EPFL Lausanne, Switzerland, he has always been a great source of help, motivation, encouragement, ideas and suggestions.

Chapter 2 of this dissertation is based on a paper [41] co-authored with Prof Luca de Alfaro, Prof Tom Henzinger, and Dr Freddy Mang; presented at CAV 2002; copyright held by Springer-Verlag Berlin Heidelberg[1], 2002. Chapter 3 is based on a paper [25] co-authored with Prof Dirk Beyer and Prof Tom Henzinger, presented at WWW 2005, published by ACM[2], copyright held by the International World Wide Web Conference Committee (IW3C2), 2005; the EPFL Technical Report Number

[1] http://www.springerlink.com
[2] http://doi.acm.org/10.1145/1060745.1060770

MTC-REPORT-2007-002 [26][3] co-authored with Prof Dirk Beyer and Prof Tom Henzinger; and a paper [27] co-authored with Prof Dirk Beyer, Prof Tom Henzinger, and Prof Sanjit Seshia; presented at ICWS 2007, copyright held by IEEE[4], 2007. Chapter 4 is based on a paper [42] co-authored with Prof Luca de Alfaro, Prof Tom Henzinger, and Prof Mariëlle Stoelinga; presented at EMSOFT 2003; copyright held by Springer-Verlag Berlin Heidelberg[5], 2003. Chapter 5 is based on a paper [39] co-authored with Krishnendu Chatterjee, Prof Tom Henzinger, Prof Orna Kupferman, and Prof Rupak Majumdar; presented at CHARME 2005; published by Springer; copyright held by IFIP International Federation for Information Processing[6], 2005. Chapter 6 is based on a paper [43] co-authored with Dr Patrice Godefroid; presented at EMSOFT 2006; copyright held by ACM[7], 2006; and Bell Laboratories Technical Memorandum ITD-06-46767J [44], co-authored with Dr Patrice Godefroid.

I am extremely grateful to Prof Tom Henzinger, Prof Edward Lee, Prof George Necula, and Prof Jack Silver, for having kindly been on my Qualifying Examination and Dissertation Committees and having spared so much of their extremely valuable and limited time over the past several years to give me extremely useful feedback, advice, comments, and suggestions that made this Dissertation possible.

As I've mentioned before, I had the incredible good fortune of taking some truly fascinating courses on computer science, electrical engineering, mathematics, and management of technology at Berkeley. These amazing lectures on these varied topics exposed me to new knowledge and ways of thinking I had never imagined and opened up so many new doors and windows in my mind. I am immensely grateful

[3] A preliminary version of this Technical Report was presented at the First International Workshop on Foundations of Interface Technologies (FIT), held on August 21, 2005, in San Francisco, CA.

[4] `http://doi.ieeecomputersociety.org/10.1109/ICWS.2007.32`

[5] `http://www.springerlink.com`

[6] `http://www.springerlink.com`

[7] `http://doi.acm.org/10.1145/1176887.1176925`

to Prof Alex Aiken, Prof Tom Henzinger, Prof Edward Lee, Prof George Necula, Prof Jack Silver, and Prof Alberto Sangiovanni-Vincentelli, for teaching me everything I know about model theory, model-checking, theorem-proving, programming language theory, design and implementation of programming languages, specification and modeling of reactive real-time systems. and design of embedded systems. I benefited beyond measure from attending amazingly insightful and thought-provoking lectures on diverse topics such as algorithms, complexity, seminal works in computer science, compiler optimizations, computer security, cryptography. computer architecture, search engines, and combinatorial games, by Prof Elwyn Berlekamp, Prof Susan L. Graham, Prof Marti Hearst, Prof John Kubiatowicz Prof Christos Papadimitriou, Prof Satish Rao, Prof Luca Trevisan, and Prof David Wagner. I had the immensely good fortune of attending some amazing classes on entrepreneurship in the high-technology industry and management of technology taught by Prof Andrew Isaacs, Prof Kurt Keutzer, Prof David Messerschmitt, Prof Reza Moazzami, and Prof Carl Shapiro. I was also fortunate to attend incredibly interesting lectures by Prof Leo Harrington, Dr Jean Paul Jacob, and Prof Umesh Vazirani, that broadened my horizons. It was truly an amazing privilege for me to have gotten the benefit of attending these fascinating courses taught by the best researchers and practitioners of each field, at the number one University on this planet.

Before coming to Berkeley, as an undergraduate student of Computer Science and Engineering at Indian Institute of Technology Kharagpur, I was extremely fortunate to have had the opportunity to work with Prof Partha Pratim Chakrabarti and Prof Pallab Dasgupta on branching-time temporal logics for property-specification and model-checking for open hardware systems. I am deeply grateful for the help and

guidance I received from them that helped me absorb the knowledge they led me to, and that I continue to find extremely valuable.

As mentioned above, in addition to an incredibly immense amount of help in all respects, my advisor Prof Henzinger provided me with constant financial support throughout my graduate studies at Berkeley. In addition, during the 2002-2003 academic year my non-resident tuition was paid by the Earle C. Anthony Fund, through the Graduate Division's Fellowship Office. I am very grateful for this support.

During my graduate studies at Berkeley I had the immensely great fortune of being selected for summer research internships at Microsoft Research in Redmond, WA; NEC Laboratories America in Princeton, NJ; and Bell Laboratories in Lisle, IL. During these internships I benefited immensely from being exposed to the fascinating research projects being undertaken by these prestigious laboratories. It was a great privilege for me to get these opportunities to interact a lot with and learn a great deal from the incredibly inspiring mentors I had: Dr Thomas Ball, Dr Michael Benedikt, Dr Glenn Bruns, Dr Robert DeLine, Dr Manuel Fähndrich, Dr Patrice Godefroid, Dr Aarti Gupta, Dr Richard Hull, Dr Franjo Ivančić, Dr Nils Klarlund, Dr James Larus, Dr K. Rustan M. Leino, Dr Sriram Rajamani, and Dr Jakob Rehof. I am deeply grateful for the great deal of time and attention they kindly gave me from which I benefited so much. I'm also very grateful for the financial support I received during these internship appointments.

I learnt a great deal and benefited greatly from the opportunity I got to attend the extremely prestigious 2004 Marktoberdorf Summer School on *Engineering Theories of Software Intensive Systems* organized by the NATO Science Committee and the Institut für Informatik, Technische Universität München, Germany, in July-August 2004. I am very grateful to my advisor Prof Tom Henzinger for kindly recommending

me for selection at this extremely prestigious summer school, and to the Marktoberdorf summer school organizers for kindly accepting me and kindly providing me with financial support to attend this school. I benefited beyond measure from the incredibly inspiring interactions I had with some of the other world-famous researchers, Dr Thomas Ball, Prof Manfred Broy, Prof David Harel, Sir C. A. R. Hoare, Prof Bertrand Meyer, Prof Jayadev Misra, Prof J Strother Moore, Prof Amir Pnueli, and Prof Mooly Sagiv, who taught at this summer school.

I was very fortunate to get opportunities to interact with inspiring researchers like Prof Rajeev Alur, Prof Rastislav Bodik, Prof Tevfik Bultan, Dr David Gay, Dr Monika Henzinger, Prof Kim Larsen, Prof Jens Palsberg, and Dr Henny Sipma, from whom I learnt so much. I would also like to thank Prof Samik Basu, Dr Fausto Bernardini, Dr Marat Boshernitsan, Christopher Brooks, Prof Luca Carloni, Prof Hao Chen, Prof Jeff Foster, Dr Sumit Gulwani. Dr Ben Horowitz, Prof Ranjit Jhala, Prof Marcin Jurdziński, Dr Sri Kumar Kanajan, Prof Christoph Kirsch, Prof Farinaz Koushanfar, Prof Viktor Kuncak, Prof Orna Kupferman, Prof Ben Liblit, Prof Rupak Majumdar, Dr Freddy Mang, Dr Scott McPeak, Prof Marius Minea, Dr Stephen Neuendorffer, Dr Jyotishman Pathak, Dr Claudio Pinello, Dr Nir Piterman, Dr Shaz Qadeer, Dr Sriram Rajamani, Dr Andrey Rybalchenko, Dr Sriram Sankaranarayanan, Dr Marco Sanvido, Dr Marco Sgroi, Prof Zhendong Su, Dr Grégoire Sutre, Dr Michael Theobald, Prof Westley Weimer, Dr Yuhong Xiong, and Dr Liang-Jie Zhang, for many immensely helpful discussions, and the valuable help and advice they have always given me whenever I have needed their help.

I would also like to warmly thank Bor-Yuh Evan Chang, Sourav Chatterji, Adam Chlipala, Jeremy Condit, Simon Goldsmith, Matt Harren, John Kodumal, David Mandelin, Bill McCloskey, Vinayak Prabhu, Armando Solar-Lezama, AJ Shankar,

Manu Sridharan, Tachio Terauchi, and Daniel Wilkerson, for many fascinating discussions, and a lot of extremely valuable and helpful feedback, comments, and suggestions on draft papers and practice talks.

I would like to thank my office-mates Krishnendu Chatterjee and Slobodan Matic for many interesting discussions, and for having given me so much valuable feedback on draft papers and practice talks. I am very grateful to Adam Cataldo, Satrajit Chatterjee, Elaine Cheong, Jike Chong, Abhijit Davare, Douglas Densmore, Shauki Elassaad, Arkadeb Ghosal, Animesh Kumar, Eleftherios Matsikoudis, Trevor Meyerowitz, Alessandro Pinto, William Plishker, Vinayak Prabhu, N. R. Satish, Haiyang Zheng, Wei Zheng, Rachel Zhou, and Qi Zhu, for many interesting discussions, and for having been part of a fascinating and dynamic research environment at the Donald O. Pederson Center that has always been an immense joy to work in.

When I first arrived in Berkeley in 2001 from the other side of the planet I was extremely lucky to have friends like Antar Bandyopadhyay, Sourav Chatterji, Arkadeb Ghosal, Rajarshi Gupta, Animesh Kumar, Rupak Majumdar, Arnab Nilim, Shivani Saxena, and Rahul Shah, who helped me so much.

Apart from them, having friends like Chandana Achanta, Smita Agrawal, Karl Batten-Bowman, Tathagata Basak, Bor-Yuh Evan Chang, Krishnendu Chatterjee, Satrajit Chatterjee, Karl Chen, Mahendra Chhabra, Adam Chlipala, Abhishek Das, Mohan Vamsi Dunga, Joy Dutta, Soumitra Ghosh, Tandra Ghose, Ajay Gulati, Anindita Gupta, Sujoy Gupta, Sanjeev Kohli, Pooja Mishra, Sridhar Mubaraq Mishra, Neelita Mopati, Ananya Nanda, Animesh Nandi, Rumana Rahman-Nilim, Amish Patel, Rabin Patra, Biswo Nath Poudel, Vinayak Prabhu, Shariq Rizvi, Kaushik Ravindran, Sourav Saha, N. R. Satish, Atul Singh, Afshan Shaikh, and Sameer Vermani made my stay in Berkeley a most enjoyable and memorable experience.

I am also very grateful for the many other truly wonderful friends I have had the delight of interacting with over the last six years, and whose friendship I will cherish forever: Dan Barak, Ushnish Basu, Shankar Bhamidi, DeLynn Bettencourt, Kamalika Chaudhuri, Tim Chevalier, Jike Chong, Alex Elium, David Farris, Deboshmita Ghosh, Eric Grismer, Gautam Gupta, Jana Van Greunen, Nili Ifergan, Jenny Ifft, Himanshu Jain, Monica Jain, Susmit Jha, Ranjit Jhala, Sudeep Juvekar, Pankaj Kalra, Nicole Kim, N. Vinay Krishnan, Manjunath Krishnapur, Shuchi Kulkarni, Shail Kumar, Ruy Ley-Wild, Rhishikesh Limaye, Abhik Majumdar, Slobodan Matic, Diana Michalek, Debananda Misra, Saayan Mitra, Marghoob Mohiyuddin, Irena Nadjakova, Mayur Naik, Matt Parker, Anjana Mitra-Parker, Bipin Rajendran, Digvijay Raorane, Joy Sarkar, Rabita Sarker, Sushil Shetty, Abhishek Somani, Lakshminarayan Subramanian, Balmiki Sur, Sumana Sur, Mana Taghdiri, Muralidhar Talupur, Darren Thornton, Rahul Tandra, Ambuj Tewari, Varadarajan Vidya, Daniel Wilkerson, Aleksandr Zaks, and John Zhu: thank you!

I am extremely grateful to the awesome staff members of the Department of Electrical Engineering and Computer Sciences (EECS), and the Engineering Research Support Organization (ERSO) at the University of California at Berkeley; and the Models and Theory of Computation (MTC) Laboratory at École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, who helped me so much and on so many occasions over the last six years: Mary Byrnes, Susan Gardner, Ruth Gjerde, Charlotte Jones, Brad Krebs, Peggy Lau, Phil Loarie, Marvin Motley, La Shana Porlaris, Fabien Salvi, Mary Margaret Sprinkle, Mary Stewart, and Sylvie Vaucher: thanks a lot !

Finally, I would like to thank my dear father Arya Kumar Chakrabarti, my dear mother Minati Chakrabarti, and my younger brother Arunava Chakrabarti, for having

been there for me whenever I needed them, and for having been a constant source of love, support, strength, and sustenance for me. Without them, I would be nothing.

Chapter 1

Introduction

1.1 Compositional design of systems

The complexity of hardware and software systems designed today calls for enhanced support for compositionality in the design and implementation process. For instance, let us consider a top-level architect of a system s. The designer partitions a complex design into high-level specifications s_i for components that are handed off to individual teams for implementation. The partitioning process may be repeated hierarchically. Each of these components make certain assumptions and guarantees about the behavior of the components they interact with; they follow certain protocols for such interactions; this information needs to be taken into account in the design and implementation process for each component. Each of these open components needs to be verified to meet its specifications. When the implementations c_i are ready, it needs to be checked that they are indeed consistent with the high-level component specifications s_i that had been the starting point. Even if that is indeed the case, it

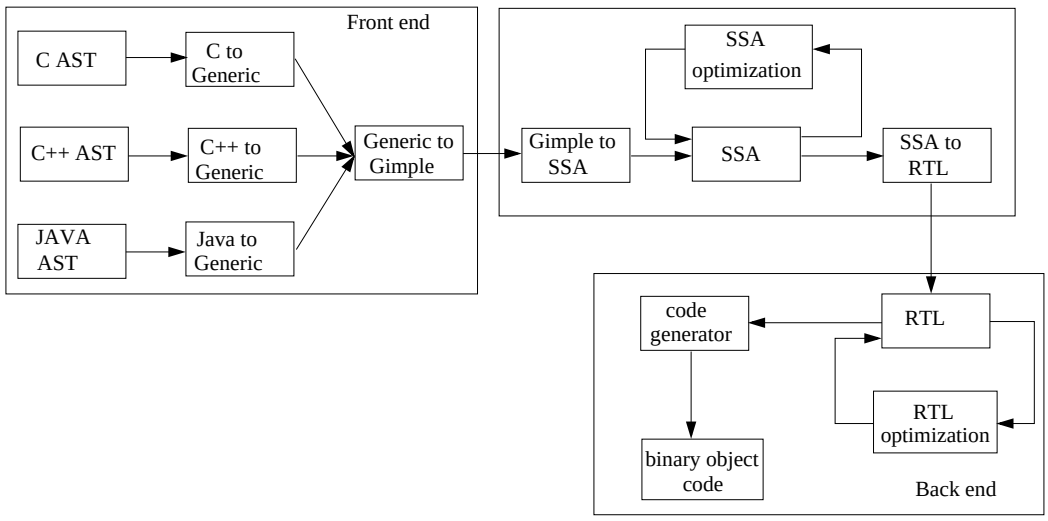


Figure 1.1. Architecture of a compiler

still needs to be verified that the combination c of the implementations c_i is a valid implementation of the original specification s for the entire system.

The formalism presented in [54, 55] provides a framework for expressing the *requirements* a component imposes on its environment, and the *guarantees* it makes if those requirements are met. This information is encapsulated in a formal model known as the *interface* of the component. The interface exposes that and only that information that the environment needs to know in order to interact successfully with the component. It is not simply an abstraction of the component; it is a specification that the environment of the component needs to satisfy in order to interact successfully with it.

Example 1.1 (Designing a compiler) Figure 1.1 presents a high-level architecture for the commonly-used GNU Compiler Collection (`gcc`)[1]. The compiler accepts programs written in languages like C, C++, or Java (among others), and translates them into binary object code [12].

[1] `http://gcc.gnu.org/`

At a high-level, the compiler consists of three parts: a "front end" that handles the various kinds of inputs, a part in between that performs machine-independent optimizations, and a "back end" that performs lower-level optimizations, some of which are machine-dependent (i.e. depending on the target hardware architecture the code is being compiled for), and finally generates machine code.

The "C AST", "C++ AST", and "Java AST" modules comprise data structures and functionality to create, represent, and manipulate abstract syntax trees (ASTs) for programs in these respective languages. These modules are also responsible for detecting syntax and parse errors in the input code. The "C to Generic", "C++ to Generic", and "Java to Generic" modules respectively comprise software to translate ASTs for these respective languages into a unified generic AST representation that can handle programs in all programming languages `gcc` takes as input. The "Generic to Gimple" module translates ASTs in this generic super-language into a simplified and structurally restricted AST format (called Gimple) that can still handle all input programs, but is a lot simpler and therefore easier to handle. These modules together comprise the "front end" of the compiler.

The next part of the compiler performs high-level machine-independent program optimizations. The Gimple format is translated into Static Single Assignment (SSA) form in which a new version of the variable name is generated for each time a variable is written to. Using this format, the compiler makes one or more passes through the SSA AST, performing optimizations such as constant folding, constant propagation, algebraic simplifications, common sub-expression elimination, loop-invariant code motion, partial redundancy elimination, etc. to perform computations that may be completed at compile time, and remove redundant computations. Finally, the code is translated from SSA form to Register Transfer Language (RTL) which can be

thought of as a machine-independent assembly language. Many low-level optimizations (e.g. register allocation, peephole optimizations, etc.) are then performed on the RTL representation. Finally, the code in RTL form is passed along to the "code generator" module which generates the best code it can for the specific hardware configuration it is written for.

With about 1 million lines of code, `gcc` is an example of a pretty large piece of software. Building systems this large (or larger, like the Microsoft Windows operating system with reportedly over 50 million lines of code) is practically impossible without a compositional approach. Modern object-oriented programming languages like C++ and Java with features like data hiding, polymorphism, inheritance, etc., allow developers to develop reasonably self-contained pieces of software that have very thin interfaces (i.e. do not expose a lot of internal implementation details, but provide guidance about how the component should be used) with the rest of the components they interact with. However, there is no formal support available to manage the information represented by these thin interfaces. Every library of components (e.g. the Java library[2], the C++ Standard Template Library[3], components like mySQL[4], etc.) therefore comes with a thick programmers' manual written in an ambiguous informal human language like English. The same is true for components of a large system like `gcc`; there is an immense body of English-language material (design documents) a programmer needs to read in order to successfully use the `gcc` components as a `gcc` contributor. As systems evolve, these design documents get edited, and inaccuracies, ambiguities, and contradictions are introduced inadvertently. A formal language

[2]Java 2 Platform Standard Edition 5.0 Application Programming Interface (API) is available at `http://java.sun.com/j2se/1.5.0/docs/api/`

[3]C++ Standard Template Library Programmer's Guide: `http://www.sgi.com/tech/stl/`

[4]`http://www.mysql.com/`

that represents the interface information in machine-readable form can thus be very helpful in improving developer productivity and improving software quality. ∎

Conventional monolithic software verification and analysis approaches are very poor in dealing with systems of such immensely large size as the `gcc` compiler mentioned above, due to the state-space explosion problem. Furthermore, as described in the example below, there are situations, such as in the realm of web-services based distributed software, where monolithic software verification is impossible to perform simply because no one entity has access to the source or even binary code for the entire system. For such situations, a compositional approach towards analysis, in which small parts of the overall system are analyzed at a time, and the overall result obtained by incrementally putting together the results of these individual smaller analyses questions, is thus very useful. However, in order to be able to do that, the analyses need to be able to deal with open systems. Conventional verification approaches have been extended to address this issue, but such approaches have usually depended upon modeling the environment of an open system and thus converting it into a closed system. This is a very labor-intensive solution. Instead, following the framework presented by de Alfaro and Henzinger in [54], we provide approaches in this dissertation to deal with verification questions for open systems without having to explicitly model the environment.

Example 1.2 (Developing an online shopping solution) In Chapter 3 we present in detail a formalism to represent and manipulate the behavior of distributed concurrent message-passing programs interacting through the web services framework. There, we shall be using a supply chain management solution for online shopping shown in Figure 1.2. It shows a "`Client`" application (which could be the end-user's web-browser) interacting with a "`Shop`" service that sells items held in inventory in a

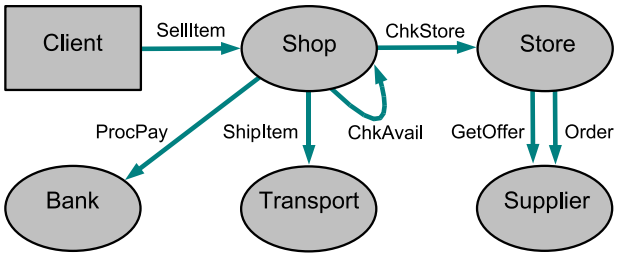


Figure 1.2. Online shopping supply chain management system

"`Store`". Items are shipped to the user using a "`Transport`" service, and paid for by the user using a "`Bank`". Items are supplied to the "`Store`" by a "`Supplier`" when inventory levels in the store run low.

Over the last decade the focus of the software development industry has shifted from development of stand-alone applications developed in its totality by one organization or entity (e.g. the Windows operating system developed completely by Microsoft Corporation, or the GNU/Linux system developed in its entirety by the GNU open source community) to the development of large-scale distributed web-based systems that are built from components developed, maintained, and run by separate administrative/commercial entities.

Thus, the "`Shop`" service above may be provided by `Amazon.com`, the "`Transport`" service by Federal Express, UPS, or USPS, and the "`Bank`" service by Bank of America. In such a situation, conventional monolithic software verification and analysis approaches (approaches that are based on analyzing the entire source or binary code for an application or system) are not applicable as no one entity has access to the entire source or even binary code of such distributed component-based systems. The approaches presented in the rest of this dissertation are geared towards addressing problems in this type of scenarios as well. ∎

1.1.1 Compatibility and Composition

Interfaces can be *composed*, and the composition is an interface that exposes only that information that is relevant to the environment of the composite interface. For example, let us consider two components A and B of a system S. Components A and B interact with each other, and also with the other components that are part of S. Component A is thus a part of the environment that B is intended to function in, and vice versa. Thus, when A and B are combined, for each of them a part of their respective environments become fixed. At this point, the composition operation involves checking if the requirements that A imposes on its environment are indeed compatible with the behavior of the part of A's environment represented by B, and vice versa. If the answer is yes, then it means that the composition of A and B is usable in some context, namely with an environment that satisfies the remainder of the combined environment requirements of A and B. Furthermore, the composite interface now needs to expose only this latter information; the information regarding environments requirements of A and B that have been mutually satisfied by each other can be forgotten.

If a mismatch exists between the mutual assumptions and guarantees of A and B, then there are two possibilities:

1. The environment of the composition of A and B may be able to take some appropriate sequence of actions that removes the mismatch. This is analogous to protocol conversion to allow interaction between two components that use different protocols. In this case, the most general behavior the environment must exhibit in order to remove the mismatch is computed and this gives the

most general (least restrictive) constraints imposed by the combination of A and B on their environment in order to allow A and B to function.

2. There may be no appropriate sequence of actions the environment can take that would remove the mismatch. Then the interfaces A and B are not compatible and their composition is the empty interface. This means that the combination of A and B is unusable in every context.

Formally, a "mismatch" is the violation of a *safety property* required of the composition of A and B together with the environment, and the problem is modeled as a two-player game: the player *System* aims to reach *error states* that correspond to exhibiting the "mismatch", and the player *Environment* aims to make such error states unreachable. The state space on which the game is played may or may not be finite. The appropriate sequence of actions taken by the environment is the latter's *strategy* to win the game.

1.1.2 Refinement

Informally, the process of system design involves starting with an abstract system description and progressively *refining* the system and component descriptions by ruling out more and more behaviors until a description allowing exactly the desired set of behaviors is obtained. The methodology of [54, 55] allows formal models of systems and components that can be refined, and a refinement relation that can be checked algorithmically. The refinement relation is a *pre-order*, i.e., it is reflexive (every interface is a refinement of itself) and transitive (if interface A refines B and B refines C then A refines C). Furthermore, the formalism ensures that a refinement A' of a component A can be safely substituted in place of A *in any context*; any mismatch

that did not occur when A was used in that context, would not occur if A' is instead. Equivalently, if two interfaces A and B are compatible, then any refinements A' and B' of A and B respectively would also be compatible, and the composition of A' and B' would be a refinement of that of A and B.

1.1.3 Specifications

Conventional software verification techniques such as model-checking only allow verification of closed systems. However, while developing a large system, often one wants to be able to find bugs earlier in the design cycle, before all the components in the entire system have been completely implemented. To achieve this, we ask the question: "given the system components we already have, is it possible to complete implementing the rest of the system such that the undesirable behavior can be ruled out?" In other words, if a bug exists in a open component c such that no strategy exists for the environment of c to compensate for it and avoid the buggy behavior being exhibited, then we conclude that the component c is defective; otherwise we conclude that the component c is acceptable, and compute the behavioral constraints the environment must satisfy in order to make the component c work correctly. This two-player game approach – viewing the verification task as a game between two players, the open system component that tries to exhibit buggy behavior, and the environment (the rest of the system) it interacts with, that tries to prevent buggy behavior from being exhibited – allows us to perform formal verification of such open systems.

This formalism allows the designer to specify safety properties of interest for an open system component, and formally verify that the component satisfies them. Specification-checking for an open system is achieved by solving two-player reacha-

bility games as before. The formalism ensures that specifications are preserved by refinement; in other words, any specification ϕ that is satisfied by an interface A is also satisfied by any refinement A' of A.

1.1.4 Application Domains

We investigated a number of application domains for interface-based design and analysis of open system components.

Hardware component interfaces

We present two game-based system modeling frameworks obtained by extending the Interface Automaton model [54]. While the Interface Automaton model provides a set of states and transitions between them labeled with input, output, or internal actions, our new model provides a richer framework involving sets of input and output variables constrained by transition predicates. This allows more concise representation of complicated systems. Furthermore, while the input assumptions and output guarantees allowed by the Interface Automaton model involve only the presence or absence of individual input or output actions, the new model allows input assumptions and output guarantees involving arbitrary predicates on the input and output variables.

The first model, *Moore interfaces* extends the standard transition relation-based model for representing synchronous systems with a symmetrically-defined transition relation for the inputs that specifies input transitions that are acceptable. The second model, *bidirectional interfaces* is obtained by modifying the previous model to allow

each component to have bidirectional connections through which inputs and outputs are exchanged between the components.

These interface models are useful for describing the input assumptions and output behavior of hardware components communicating through boolean signals on uni- or bi-directional wires. Composition and compatibility-checking are achieved through solving two-player finite-state reachability games. Refinement-checking is achieved through checking alternating simulation. Properties of the PCI Bus and Token Ring protocols are verified in terms of compatibility- and refinement-checking in this formalism [41].

Recursive software components

We build on the finite state Interface Automaton model [54] to obtain a pushdown model that can be used for representing *method availability constraints* (pre-conditions and invariants that must be satisfied when a method is invoked) of single-threaded recursive open software components. This framework thus allows for formal representation of high-level inter-component interaction protocols (in terms of method-call ordering patterns) for software components. Composition and compatibility-checking is achieved through solving two-player reachability games on the configuration graph of a push-down automaton. This approach is not discussed any further in this dissertation; further details can be found in [40, 38].

Distributed asynchronous systems

We present a formalism to describe the behavior of recursive and multi-threaded distributed open components communicating with each other in the web-services

based application-development setting. In this scenario formal behavioral specification techniques are of crucial importance as applications usually cross vendor, administrative, or other boundaries across which neither binary nor source code can be shared. Our formalism offers explicit constructs to model recursion and concurrency (including dynamic thread creation) in ways that are natural in this programming context. Algorithms are provided to check if two or more interfaces are compatible; if a web service can be safely substituted in place of another in any arbitrary context; and if a given web service satisfies a given temporal safety specification representing a desired behavioral property [25, 26, 27].

Non-functional properties

The notion of incompatibility is modified; in contrast to the formalism in [54] where a configuration is illegal if it corresponds to an interface producing an output that a peer interface refuses to accept, in this formalism we consider finite-state interfaces with integer labels representing instantaneous consumption (positive or negative) of resources on states. Positive resource consumption may correspond to buffer allocation; negative consumption may correspond to deallocation, for example. A configuration is illegal if the node labels seen so far fail to satisfy a given predicate. Two kinds of predicates are considered: *node limit* predicates correspond to the instantaneous consumption being above a critical threshold; and *path limit* predicates correspond to the cumulative consumption of the resource being above a critical threshold. The formalism also allows the designer to specify liveness conditions such as Büchi constraints.

Two kinds of problems are considered: the *strategy synthesis* problem corresponds to discovering ways to use a system to keep the instantaneous or cumulative consump-

tion of a scarce resource under a given threshold; and the *resource synthesis* problem corresponds to computing the minimal amount of a scarce resource that must be given to the system such that such a strategy to use it may exist. Both problems can be solved algorithmically in an efficient manner [42].

Quantitative Verification

We further generalize the above framework into a *quantitative* theory of verification that generalizes the conventional boolean framework of μ-calculus model-checking. Thus we have *quantitative* structures that are graphs with finitely many vertices, but with every vertex labeled by a set of *quantitative* propositions, each taking an integer value. A *quantitative* automaton maps an infinite path in a quantitative structure to an integer. For practical examples, quantitative bound functions exist that allow decidability of model-checking in this setting [39].

Software architecture extraction by analysis of information shared between components

We apply static analysis to estimate the amount of information represented by the interfaces between software components. In a C codebase, each elementary component is a C function that communicates with other components by receiving calls and arguments, calling functions and passing parameters, sharing globals, etc. We investigate techniques to automatically estimate the amount of information represented by these inter-component communications. Components that share a lot of information are considered more closely coupled than components that share little information. We consider two kinds of communications: represented by control-flow and data-flow respectively. These are captured by the control and data interfaces of C functions.

We show that using control interfaces alone, it is possible to partition the codebase into meaningful *units* that may be tested effectively in isolation [44, 43].

1.2 Related Work

A large number of promising approaches have been proposed for automated support for software design, analysis, testing, and verification. Broadly, they may be classified into approaches that work statically (at compile time), and approaches that work dynamically (at run time).

1.2.1 Static approaches

Type systems

Type systems are a commonly used framework for compositional specification and analysis of system behavior. A *type* is simply a set of values; e.g. the type `int` used in the C Programming Language represents the set of all integer values that can be represented in 32 or 64 bits (depending on the machine and architecture for which the C compiler in question has been written). Similarly, the type `boolean` in Java represents the set of values `true` and `false`. Since all data manipulated by a program in a computer is stored as sequences of binary digits, the type associated with a *variable* (the contents of a portion of memory) determines what operations (addition, concatenation, negation, etc) make sense for, and are hence allowed on, that data. *Type-checking* is a *semantic analysis* done by a compiler to make sure that all operations occurring in the program are *type-safe*, i.e., they occur on data that allows said operation. Languages can be weakly-typed or strongly-typed; weakly-

typed languages (e.g. C++, Perl[5], JavaScript) allow large numbers of implicit type conversions in which the compiler allow data of one type to be treated as another. Strongly-typed languages (like Java) do not allow this and force greater discipline in programming on the part of the developer. While there exist languages like assembly language (and even high-level languages like some varieties of Forth[6]) that are untyped, the programming language community has been, over the last several decades, steadily moving towards more and more strongly typed languages in recognition of the fact that type systems help detect large classes of program errors at compile time. In addition to the type-checking approach used by standard programming languages like C, C++, or Java, there exist languages like ML, OCaml[7], and Haskell[8], which allow *type inference*, in which the compiler frees the programmer from the chore of having to manually declare the type of each variable used, and wherever possible automatically guesses the types of various variables used in the program by analysing the operations performed on them, and complains with error messages only when it detects a true incompatibility. Most commonly-used programming languages like C, C++, C#, or Java are *statically typed*: i.e. the compiler associates types with variables and expressions used in the program at compile time. In contrast there also exist *dynamically typed* languages like Ruby[9], Lisp[10], JavaScript, Python[11], etc., in which the compiler associates types with values at run time only, and no types are associated with variables or expressions at compile time. While this allows greater flexibility by letting the same variable to refer to values of different types at different points of program execution, this approach also means that all type-mismatch errors

[5] `http://www.perl.org/`
[6] `http://www.forth.org/`
[7] `http://caml.inria.fr/ocaml/`
[8] `http://www.haskell.org/`
[9] `http://www.ruby-lang.org`
[10] `http://www.lisp.org`
[11] `http://www.python.org/`

can only be discovered at run time. Due to this drawback, dynamic typing is not popular in mainstream programming languages used to develop extremely large and complex systems.

Viewed in the context above, the work presented in this dissertation can be seen as an extension of conventional type systems into representing richer information (assumptions and guarantees about high-level behavior) about the behavior of more diverse sets of program elements (such as components, or large portions of software, as opposed to the behavior of just individual variables, structures, unions, or objects in conventional type systems). The concept of *type safety* is analogous to the notion of *interface compatibility* presented in this dissertation. As in strongly-typed languages that do not do type-inference (such as Java) and thereby require the developer to explicitly annotate variables with type declarations, in our approach there is a similar overhead placed on the developer in the form of the requirement to provide behavioral assumption and guarantee information by writing interface descriptions for components. Furthermore, just as type-inference (as in ML, OCaml, or Haskell) reduces this annotation overhead on the developer by automatically extracting large parts of the type information that would otherwise have been asked of the developer, our framework is similarly capable of being improved upon through automated *interface extraction* that would similarly reduce the annotation overhead on the developer. There has been very interesting work on this latter topic independently by Henzinger et. al. [92] and Alur et. al. [21].

Type systems allow semantic analysis of open systems. A part of a program (say a set of functions and variable declarations in a language like C or C++) can be analysed by the compiler and checked for type mismatch errors just as well as a complete system can. This is in contrast with conventional model-checking based

software verification techniques, which can only be used when the entire system description is available (or a partial specification is "closed" by adding stubs comprising rudimentary definitions for the parts of the system that are unavailable; a painstaking and burdensome process that often places an unacceptably high overhead on the verification team). Compilers for type-inference based programming languages such as OCaml, when used to separately compile OCaml program fragments, compute the most general types for each variable based on the constraints generated from the type information in the program fragments given. This is analogous to our *composition* operation for interfaces, wherein we compute the most general environment behavior allowed by the constraints generated from the behavioral assumption and guarantee information in the interfaces for given system components.

An important concept in type theory is *subtyping*: a *subtype* is a data-type that may be substituted in place of its *supertype* in any context where the supertype may be used. For example, if a function takes a single argument that is required to be an integer, it would be perfectly fine to invoke the function on an even integer. Any operation permitted on an integer in general is also permitted on an even integer. This concept of *substitutivity* or *refinement* takes a central place in our approach towards compositional system design: we define precisely when an interface may be substituted in place of another in an arbitrary context. This allows flexibility in the design process; e.g. in a web-services based distributed computing scenario, it may be possible to check if the behavior provided by one shipping service provider such as Federal Express can be replaced with another, such as USPS, in any arbitrary context.

Thus, our approach which combines the benefits of model-checking based approaches with the ability to handle open systems similarly to type systems, thus can

be seen as bridging the gap between these two complementary software verification approaches.

1. Type qualifiers: This approach [73] allows augmentation of the built-in static type system of a statically-typed programming language like C with more refined static types using extra user-defined *type qualifiers*. For example, a qualifier such as `tainted` can be used to mark untrusted data obtained by the program from untrusted and adversarial users who may be trying to crash the system. Then, the possible paths along which the untrusted data items can propagate through the program can be tracked statically through the type-inference and type-checking process. Then, calls to functions such as `printf` can be checked for use of `tainted` data. This approach has been used to detect format-string vulnerabilities in C programs [130]. In a fashion similar to type inference mentioned above, *qualifier inference* is used to deduce a large fraction of type qualifiers based on a few that the programmer needs to manually add to start with; this substantially reduces the annotation overhead on the developer [73]. Possible bugs are uncovered in this scheme through type mismatches. As discussed above, open systems can be type-checked in this approach, just like the design and analysis framework presented in this dissertation. However, in contrast to our approach which works on high-level abstract system descriptions, this approach works directly on qualifier-annotated source code in a programming language like C.

   Fundamentally, a type represents a set of assumptions and guarantees about the operations permitted upon (or behavior exhibited by) a variable or a function. From this viewpoint, permitting user-defined type qualifiers is equivalent to giving the user (the system designer and developer) the ability to extend the

base type system of the programming language being used, with support for expressing newer types of behavioral assumptions and guarantees that the system designers and developers deem important for that particular system. Thus, this approach may be viewed as a generalization of a conventional type system in the same general direction of allowing greater ability to represent inter-component assumptions and guarantees.

Our framework can be thought of as further generalizing this approach, as in our case types are not forced to be static; the type of an object (as represented by the set of operations allowed on that object) in our setting may change over the execution of the program, similar to the approach taken in dynamic typing. However, while dynamic-typing based type systems are not conducive to detection of type errors before run time, our framework permits model-checking based compile-time analysis of the run-time behavior of the program to detect type mismatches before the program is actually run.

2. Type systems for memory safety: Regions [79], and linear types [100, 59, 68] are two type-based methods proposed to allow compile-time guarantees of memory-safety. These methods usually impose severe restrictions on the developer with regard to how variables or objects of these types can be used. For example, objects with linear types can be used exactly once; they cannot be duplicated. Region-based memory management systems divide memory into regions; every object, when allocated, is associated with a region. Unlike in languages like C++, memory is not freed one object at a time; entire regions are deleted, deallocating every object in it in one go. This forces greater discipline on the part of the programmer with regard to management and deallocation of memory, and allows languages with such memory management systems to guarantee

memory safety (usually dynamically). As expected, both of these techniques are applicable to analysis of open systems.

Though the techniques concerned are completely different, there are fundamental similarities in motivation between these approaches towards managing access to shared data, and our Resource Interfaces formalism for managing scarce shared resources (more details about Resource Interfaces will be presented in Chapter 4). Essentially, a linearly-typed object can be thought of as a resource permitting a *node limit* of 1; i.e. permitting only one user to have access at a time. There are also other application scenarios where access to scarce but non-unique objects (e.g. database handles, of which there may be a small number, such as 50, but many more than just 1) need to be managed in a fully programmatic and software setting, and the use of linear types (because they insist on keeping the concurrent access limit (*node limit*) at 1) might be difficult. While reference-counting based dynamic solutions have been proposed to address problems of this type, it seems an interesting future problem to investigate ways to combine the two dramatically different classes of techniques used in the linear types literature with the model-checking based methods used in Resource Interfaces in order to obtain better compile-time solutions to this problem.

3. System-level types: The Ptolemy II [3] component-based system design environment uses *system-level types* [106] to characterize runtime interactions between components. The components interact by calling methods implemented by other components. The type of a component in this setting is an interface automaton that restricts the sequences of method calls that are allowed by the component. A call to a method m implemented by a component A is an input to the inter-

face automaton representing A; a call from A to a method m' implemented by a different component B is an output from the interface automaton representing A. Type-checking different interacting components for compatibility involves composing the interface automata representing the behavior of the components concerned. Subtyping relationships between components correspond to refinement relationships between the corresponding type automata. The components are organized into a lattice reflecting their subtyping relationships; this allows easier design of polymorphic components and simplifies type-checking in the Ptolemy II system. In addition to these static checks, the Ptolemy II system also supports enhanced debugging functionalities by implementing reflection of component states and runtime type checking to find incompatibilities that correspond to mismatches in inter-component interaction protocols in complex systems with large numbers of components.

This work is closely connected to the work presented in the rest of this dissertation in the fundamental view of interfaces as behavioral types. While this work demonstrates the practical applicability of an interface-automata based behavioral type system for efficient support for compositional system design in the Ptolemy II setting, in the rest of this dissertation we present other interface formalisms that are useful in other domains, such as distributed asynchronous services, or in the context of resource-constrained embedded software.

4. Session types for concurrent distributed protocols: *Process calculi*, such as π-calculus, are often used to represent the behavior of concurrent distributed programs. Such representations usually focus on the communication behavior of the interacting components of the concurrent program, by describing the sequences of *messages* sent over *channels* between the interacting peers. Mes-

sages can be given types, and channels can be restricted to be able to carry messages of certain types only. Other extensions have been proposed in which certain channels are allowed to carry messages of different types at different points of program execution. Such a channel is then characterized by a *session type*, which is a sequence of message types possibly including branching choice points. Session types thus represent sets of sequences of types of messages that may be sent or received over the communication channel. Such a formalism can be used to describe finite-state message exchange patterns for distributed or local peers communicating with each other over communication channels [122]. They have been used to describe distributed client-server interaction protocols such as POP3 [80] and SMTP [122]. Though the techniques involved are completely different, this approach is similar to ours in that both are intended to formally characterize the behavior of interacting open system components, and reason about their compatibility in the presence of a helpful environment. However, while this approach focuses on finite-state interaction protocols, the approach presented in this dissertation allows representation and analysis of unbounded-state interaction protocols [40, 38, 25, 26, 27]. More details about an unbounded-state model for representing the behavior of distributed asynchronous services will be presented in Chapter 3.

5. Typestates: In contrast to the standard type system framework where an object has the same type throughout the execution of the program, *typestate systems* [131] have been proposed as a generalization allowing the state of an object to change over time as a result of program actions. As a result, a typestate system is able to statically enforce various safety properties by making sure that certain program operations are never allowed to happen on objects in certain

typestates. One common problem plaguing typestate systems is the *aliasing* problem: if the same object is referred to using different references (*aliases*) in different program operations, the typestate system is in general unable to resolve all aliases of a given object to a canonical name, and is hence in general unable to track the state of aliased objects precisely. Generalizing the standard typestate approach which assigns a single state to each object at every execution point, other models allow each object to be a member of a subset of a given set of typestate sets, thus allowing each object to be in any subset of the set of all possible typestates at any point of time [104]. These generalized approaches provide better solutions to the aliasing problem. However, while these approaches are intended to work directly on source code, we focus on behavioral descriptions at a higher, more abstract level. This allows us to require interface descriptions to be made in a language that is much simpler than a general-purpose programming language, and thereby rule out the aliasing issue by construction. In contrast to typestate systems that track system behavior at a fine-grained per-variable basis, we focus on higher-level safety properties at the inter-component interaction protocol level.

6. Behavioral subtyping: An interesting model-theoretic approach to formally define the notion of behavioral subtyping is presented in [108]. Our formalism is consistent with this approach, and the notion of behavioral subtyping in this setting is equivalent to our notion of substitutivity or refinement. While the abstract model-theoretic definitions of [108] provide a unified domain-independent definition for behavioral subtyping and take a completely abstract view of the specific syntax and semantics of the target language (and thus do not attempt to take advantage of any particular characteristics of any specific languages or

application domains in order to achieve efficient algorithms for checking behavioral sub-typing relationships in those specific languages), we approach the issue of behavioral subtyping on a domain-by-domain basis, and have different formalisms for different application domains (such as hardware components, distributed asynchronous services, or resource-constrained embedded software), and different algorithms that take advantage of the specific properties of the latter.

Static analysis, theorem-proving, model-checking, and related techniques

An alternative approach towards program verification distinct from type-systems based ones is a class of compile-time program analyses involving tracking the flow of control and data and reasoning about the effects of program statements on tracked data with various degrees of precision. This class of approaches include *static analysis*, *model checking*, and *theorem-proving* based reasoning techniques. Static analyses include syntactic checks for adherence for coding standards and best practices, analysis of the flow of control and data through the program (which may be done in a flow-insensitive, flow-sensitive, or path-sensitive manner depending on the degree of precision desired), points-to analysis (finding sets of objects or variables that a pointer can point to at various points of the lifetime of the program), shape analysis (discovery and verification of properties of dynamically-allocated linked data structures), etc. Others have proposed static-analysis based methods to allow developers to check satisfaction of specific coding rules by their software codebase [66, 86]. In contrast to the analyses above, this latter approach requires developers to provide additional information in the form of the rules they want to be checked. Extended static checking [60, 71] has been proposed to check safety properties statically using a

combination of static analysis and theorem-proving. In this approach, developers are allowed to provide additional annotations describing information about their software that they have access to while writing the program, but which do not form part of the source code they write (e.g. beliefs in their mind about the behavior of their software and relationships maintained among variables, objects, functions, methods, and other program elements); such as *pre-conditions* assumed and *post-conditions* guaranteed by various functions or methods, *invariants* maintained in various classes in an object-oriented setting, etc. This additional information is written in machine-readable annotations that the program checker reads and reasons about at compile time using built-in theorem-provers. Like our approach presented in this dissertation, these approaches are compositional and are able to gracefully handle open systems. Also, in contrast with *dynamic* techniques that will be discussed below, in order to be effective these approaches do not depend on the developer or the tester to provide exhaustive test suites that cover enough execution paths. In contrast, our approach focuses only on the inter-component behavior and is hence able to perform analysis of protocol-level interaction errors between components while completely ignoring the behavioral information completely internal to each component.

In contrast to static analysis, our approach is able to check temporal safety property specifications. This benefit is not without cost; the price we pay in our approach is the need for developer-provided interface descriptions for the components being analyzed; annotations that are not needed by static analysis techniques that run directly on the component source code, and can be labor-intensive to generate manually. Extended static checking depends upon developer-provided annotations as well.

Two different techniques for automated extraction of component interfaces from source code have been proposed [92, 21] that can substantially mitigate the overhead

cost of this need for additional information. The first of these two techniques is based on a variation of a previously-proposed interesting combination of static analysis, model-checking, and theorem-proving to automatically extract boolean abstractions of programs from source code and check safety properties [22, 23, 93, 91]. Though this latter method was geared towards abstraction (and verification) of closed systems, interesting combinations of these same techniques has been used for automated abstraction of interfaces for open systems [92]. The second approach for interface synthesis uses symbolic model checking and learning finite automata [21]. For the reason mentioned above, these two approaches for interface synthesis complement the work on interface analysis (compatibility-checking, specification-checking, and refinement-checking) presented in this dissertation.

1.2.2 Dynamic approaches

Software testing

This is the standard approach for software quality assurance that has been traditionally used in the software engineering industry for the last several decades. The effectiveness of this approach depends on the software developer or testing engineer to provide an exhaustive test-suite that covers all important program paths. This is a very hard problem, as the number of execution paths in general grows exponentially in program size (as the number of paths can double with each `if-then-else` element in a linear sequence of `if-then-else` blocks that provide a series of bifurcation and join points for control to flow through). Testing does not work in general on open systems, e.g. if unimplemented functions or methods are called by the system under test. So, *stub functions* need to be written to provide place-holder implementations

for such functions to close the system. A test harness needs to be written to exercise all implemented functions. Our approach presented in this dissertation solves all the above problems associated with testing: we can deal with open systems with unimplemented functions or methods, and the efficacy of our approach does not depend on the exhaustiveness of the test suite provided. Furthermore, while testing large systems is a very expensive problem due to the extremely large numbers of execution paths therein, our approach is fully compositional and better suited for analysis of component-based systems with large numbers of components. However, testing has certain compelling benefits that are very hard to provide in a formal software verification solution; e.g. since there are relatively few up-front costs one has to pay to start testing software (apart from writing the test-cases one by one), it is often possible to find some of the most shallow bugs (which can be quite numerous till pretty late in the development cycle) at relatively low cost with testing, compared to using a formal method that may require significantly more groundwork (e.g. interface descriptions in our case) before the benefits start to obtain. Furthermore, testing is intuitively simple and easily understood by even novice programmers, in contrast with formal approaches which currently require somewhat greater sophistication on the part of the user. For these reasons, our approach is certainly not intended to be a replacement for testing; it is intended to be complementary to it. Testing is a time-tested software quality technique which will keep its place in the software development process for a long time to come for the reasons mentioned above. Hence, in Chapter 6 we provide a method to partition large pieces of software to obtain smaller pieces which can be tested separately to get better test coverage, without leading to too many *false alarms* (possible behavioral mismatches that appear to be programming errors when seen in the limited piecemeal testing context, but which later turn out to not be errors after all when seen in the context of the entire system as a whole; such false

alarms can occur as a result of the incomplete behavioral information available during piecemeal testing). This method is based on analyzing the patterns of information sharing between different parts of the system under test.

The "Design by Contract" methodology

This is a promising dynamic approach to facilitate compositional software design and implementation proposed by Bertrand Meyer and supported by the Eiffel[12] programming language developed by him and his collaborators. In this approach, a system is developed by creating components that satisfy certain "contracts" made with each other. These contracts restrict the behavior of the components. The contracts are enforced at runtime. In Eiffel, methods are equipped with pre-conditions and post-conditions, and classes are equipped with invariants. When a method is invoked, all its pre-conditions must be true. When the method finishes execution, all its post-conditions must be true. Class invariants are established by constructor methods and must be maintained by all other methods. Eiffel is an object-oriented language supporting inheritance. Pre-conditions can only be weakened by inheritance, and post-conditions can only be strengthened. This ensures that the pre-conditions of a method implemented by a class are satisfied whenever the pre-conditions of the over-ridden method in the parent class are, and vice-versa for post-conditions. Eiffel also supports assertions and loop invariants. Some of these features (e.g. assertions) have been incorporated into other mainstream programming languages like C, C++, C#, and Java. The contracts are enforced at runtime and their violations constitute bugs. However, if a bug lies on an infrequently executed program path, then the probability of discovering it using this approach would in general be very low, as the

[12] `http://www.eiffel.com/`

approach depends on the developer/tester to provide a good test-suite that covers all important program paths [115]. Thus, while this approach vastly improves upon testing, by allowing developers to express complex conditionals and invariants that are maintained by their carefully crafted code, and makes it easier to discover bugs introduced by subsequent changes that break those invariants, the approach remains fundamentally vulnerable to the same inadequacies that characterize testing: in a large program the number of execution paths (which grows exponentially with program size) is too large and reaching one hundred percent code coverage (measured in terms of any of the standard test coverage metrics: line coverage, branch coverage, or path coverage) is extremely hard and prohibitively expensive. Furthermore, as in testing, and in contrast to our approach, runtime checking of contracts in the Design-by-Contract/Eiffel approach requires that there be no calls to unimplemented functions or methods.

The notion of behavioral "contracts" between interacting software system components is fundamentally similar to the notion of characterizing open system components with interfaces that capture the behavioral assumptions and guarantees between them. However, as mentioned above, the Eiffel approach is based on dynamic checking of contract-satisfaction at run-time. In contrast, our approach is based on compile-time model-checking to statically uncover behavioral interaction errors without depending on a developer-provided test-suite for exhaustive dynamic analysis. The Eiffel approach is also oriented towards expression of contracts at a very fine-grained source-code-variable by source-code-variable level. In contrast, our approach allows expression of behavioral "contracts" at a higher level of abstraction.

Dynamic analysis

In addition to the approaches mentioned above, several promising dynamic analysis techniques exist, some of which manage to mitigate to some extent the need for developer-provided test-suites mentioned above. However, even those techniques remain unable to handle open systems, particularly with regard to calls to unimplemented functions or methods, that the approach presented in this dissertation is able to handle. We present a brief discussion of a few salient dynamic analysis techniques as follows:

1. Race detection: This approach instruments source, or often, binary code to monitor accesses to shared memory in multi-threaded programs to detect race conditions [128]. As expected, this approach depends on availability of a good test-suite that covers enough execution paths. Furthermore, this type of approaches cannot handle open systems with calls to unimplemented functions or methods unless *stub functions* are written to close the system under analysis. In Section 3.5 we present a case study involving using our formalism and methodology for checking safety specifications for detection of a race condition in a distributed web-services based application that uses the Amazon.com E-Commerce Services (ECS) framework. It should be noted that our approach presented therein, as in the rest of the dissertation, works at compile-time, in contrast with the dynamic approach of [128]. Furthermore, while the approach of [128] works directly on source or binary application code, our approach works at the level of interface descriptions. The domain of application of these two approaches are different; while the approach of [128] focuses on checking complete programs in the monolithic setting, our approach presented in Chapter 3 is oriented towards verification of distributed applications in the web-services

setting where no one entity usually has access to the entire source of binary code of the entire web-based application.

2. Invariant discovery: This approach uses a set of test inputs given by the user to discover possible invariants [67]. Techniques of this kind can be combined with the "Design By Contract" methodology presented above to reduce development-overhead involved in manually extracting program invariants. Similarly, we hypothesize that it may be possible to use techniques of this kind to automatically extract, from component source-code, higher-level behavioral properties that might be part of the component interface. Such an approach, if feasible, could complement the methods mentioned above [92, 21]. However it should be noted that currently in this approach there is no satisfactory solution to the problem of dealing with calls to unimplemented function or method calls.

   However, invariant-discovery using this approach requires the software developer to provide a good set of inputs that will cover enough program paths. However in conjunction with a promising new technique to automatically discover test inputs that exercise new program paths as explained below, it can be used to much greater advantage.

3. Concolic execution: As mentioned above, this approach uses a combination of dynamic and static analyses and theorem-proving ("**conc**rete and symb**olic**") to automatically discover new test inputs to exercise new program paths. Coverage is usually low in the presence of complicated program operations that the theorem-prover finds difficult to reason about. However, in a large class of real-world programs the program logic is simple enough to permit automatic test input generation using this method to achieve reasonably good coverage. Furthermore, in conjunction with the invariant-discovery approaches mentioned

above it can be used to automatically discover interesting program invariants without having access to an exhaustive test-suite.

1.2.3 Design Patterns

This [77] is an interesting approach towards codifying commonly-used programming techniques. A design pattern is a template for a specific commonly-occurring problem in software engineering. Design patterns have been classified into categories based on the types of problems they address, such as creational, structural, behavioral, or concurrency patterns. Of these, behavioral patterns and concurrency patterns address the same general types of problems that are the target of our behavioral specification formalism. However, our approach focuses on formal specifications of behavior, and algorithmic techniques for checking composition, compatibility, specifications, and refinement, whereas the focus of the Design Patterns community has so far been on providing concepts that can be used by developers to efficiently communicate the inter-component interaction schemes they use; the actual use of these patterns requires human involvement throughout, and does not seem amenable to automation. Moreover, the specific types of Design Patterns recommended [77] for common use are simply guidelines that help establish a generic and dynamically reconfigurable inter-object communication architecture in a completely domain-neutral, program-neutral and functionality-neutral way; they do not deal with any particular behavioral details that may occur in any specific program or application to deliver any specific functionality in any specific application domain. For example, the Chain of Responsibility pattern recommends that developers avoid coupling a sender of requests to a specific receiver by allowing the request to be passed along by a chain of possible receiving objects until one of them handles it. The Visitor pattern is based on the idea that

descriptions of operations performed on complex data structures should be decoupled from the descriptions of the data structures themselves. Design Patterns are thus intended to codify best practices and rules of thumb that apply to practically all programs and therefore are necessarily concerned with behavior only at a very high-level of abstraction where all program-specific domain-specific and application-specific details are abstracted away. In contrast, in this dissertation we provide domain-specific formalisms to represent application-specific behavioral assumptions and guarantees made by open system components about each other in order to implement specific functionalities.

1.2.4 Software architecture specification

This [78, 13, 14, 47, 48] is an analogous approach towards codification of commonly-used software architectures. These approaches have focused on finding solutions to commonly-occurring software architecture problems involving protocols for inter-component communication, global control structure, physical distribution, scaling and performance, choice between design alternatives, etc. The scope of software architecture issues is larger than that of the design pattern issues discussed above. For example, in [78] the authors present some widely-used software architectural elements such as the "pipes and filters" abstraction that allows components to be implemented separately and decoupled from each other and combined in various ways as long as the types of inputs and outputs exchanged on the pipes match up; object-oriented design in which functionality and associated data are combined into logical entities called objects or classes, and the details of the implementation are hidden to the maximal extent possible from users; event-based programming, in which entities register with an event manager as receivers of various program events (such

as a mouse-click) and are "implicitly invoked" (by the event manager) when those events occur, thereby freeing up the developer from the worry of having to constantly keep her application software explicitly enabled to receive and appropriately handle all possible events, allowing better compartmentalization of the functionality implemented in separate components without mixing them up in an explicitly-implemented unified event manager in the application code, and thereby allowing easier addition of new components or removal of existing components from the system.

Our focus on formalizing inter-component behavioral assumptions and guarantees causes us to address some of the same issues of modeling inter-component interaction addressed by Garlan et. al. in formally defining and representing commonly-occurring architectural elements (such as the commonly-used client-server architecture) [13, 14]. However, there are certain fundamental differences in the modeling approaches used. Garlan et. al. use a model – with processes and events, named processes and recursion, sequential and parallel composition, and internal and external choice – based on Communicating Sequential Processes (CSP) [94]. While internal and external choices allow explicit representation of responsibility for, and control of, action and reaction on the component (internal choice) or the environment (external choice) and is in this regard somewhat comparable (in a severely limited sense) to the player-specific control information represented in two-player games (like what we use in our models presented in this dissertation), that is where the similarity to two-player games ends in the frameworks used in this work. In contrast, two-player games forms the fundamental basis of the models presented in this dissertation, and is the very reason that allows us to perform verification tasks (such as specification-checking) on open system components. Furthermore, in contrast to the work presented in this dissertation, the work of Garlan et. al. [78, 13, 14, 47, 48] has focused on providing conceptual frame-

works and solution templates for commonly-occurring *domain-neutral* (but detailed and expressive enough to allow representing application-specific issues, in contrast to the Design Patterns methodology discussed above) architectural problems; not on *domain-specific* formalisms to represent application-specific architectural decisions. Thus, while the CSP-based model has certain broad similarities to, for example, the *protocol interfaces* model we use to represent the behavior of distributed asynchronous services in Chapter 3 (in certain respects, e.g. ignoring the differences regarding the two-player game aspect), there are major differences (e.g. the two different kinds of parallel composition allowed by our model, a domain-specific modeling decision) motivated by the characteristics of the application domain.

These modeling differences notwithstanding, there are many commonalities in the goals afforded by the formalism of Garlan et. al. and those afforded by ours: the notion of *compatibility* (between *ports* and *roles* representing *connectors* and *components* respectively in the former, and between component interfaces in the latter) is analogous and central to both approaches. While the formalisms presented in this dissertation focus on checking *safety property specifications*, this notion is analogous and theoretically equivalent to the notion of *deadlock-freedom* in the work of Garlan et. al. (since any given pair of a system description K and a safety property p may be translated into a new system description K' that deadlocks if and only if K can generate an execution trace that violates the safety property p). In addition to checking compatibility and deadlock-freedom, Garlan et. al. have also investigated other interesting (though completely orthogonal to the direction pursued by us in the research presented this dissertation) applications of this same fundamental component-and-connector architecture model, such as detection of integration mis-

matches and automated synthesis of glue code to handle such mismatches [112]; and merging and differencing of architectural views [6].

1.2.5 Game semantics for programming languages

A lot of effort has been devoted by the programming language community on providing a syntax-independent denotational semantics for a higher-order functional programming language such as PCF (the Programming Language of Computable Functions) [11, 98, 123]. For this purpose, Abramsky et. al. model a program as a strategy for player P playing a game against an environment E [8, 7]. The moves of players P and O correspond to exchanging input and output data from the program to its context and vice versa. Functional and imperative programming language constructs such as values, functions, higher-order functions, products, function composition, mutable store, control operators, nondeterminism, subtyping, etc. are represented as strategies of player P against the player O. The two players are assigned various strategies and the set of possible outcomes of the games they play with each other defines the semantics of the language.

However, while the goal of the work by Abramsky et. al. is towards providing a new formal mathematical model of computation for explicitly modeling modern programming language concepts, in the compositional verification approach we present in this dissertation we model component-based open systems as two-player games in a similar way but for a different purpose. We model open systems as two-player games between a player System and a player Environment to verify safety properties for open systems with an optimistic view about the behavior of the environment, thus allowing us to verify complex and large systems in a compositional fashion.

In our approach the Environment wins if certain safety properties are satisfied while the game is played, and System wins if it can violate any of those safety properties. The winning strategies correspond to desired behavior of the environment and thus represent conditions under which the system component may be used. As mentioned before, we do this with the goal of representing and analyzing the behavioral assumptions and guarantees to each other by system components in order to enable compositional design of component-based systems. Thus, we are interested in specific games and in being able to compute if any outcomes correspond to violations of our safety specifications. Also, we focus on specific application domains (like hardware circuits, distributed web services, embedded software, etc) and construct specific formalisms focused on taking advantage of domain-specific opportunities for achieving easily-usable syntax, expressive semantics, and algorithmic optimizations. In contrast, Abramsky et. al. in [8, 7] define the allowed strategies of the two players but then are not concerned about the outcomes of any specific games that may occur in specific practical problems. Furthermore, their goal is to capture the semantics of an expressive programming language, independent of any simplifications or optimizations afforded by any specific application domains where that language may be used.

While Abramsky et. al. originally developed their framework with a different motivation as mentioned above, their approach is certainly amenable to being re-targeted towards a compositional framework for specification-checking for open systems, and they have done some interesting work towards that goal as well [10]. In contrast to the approaches presented in this dissertation (e.g. the unbounded-state model of *protocol interfaces* presented in Chapter 3) their work has focused on finite-state models. As before, and in contrast to our approach, their methodology is domain-neutral and thus

does not take advantage of any particular characteristics of any specific application domains.

1.3 Outline

The rest of this dissertation is organized as follows. In Chapters 2, 3, and 4 we present three interface formalisms for three different application domains: hardware systems, web-services based applications, and resource-constrained embedded software platforms respectively. Chapter 5 consists of a quantitative verification framework obtained as a generalization of the model presented in Chapter 4. In Chapter 6 we present a formalism that allows automated software partitioning based on the heuristics that estimate the amount of information represented by *control interfaces* of functions in a language like C.

Chapter 2

Synchronous and Bidirectional Component Interfaces

We present *interface models* that describe both the input assumptions of a component, and its output behavior. By enabling us to check that the input assumptions of a component are met in a design, interface models provide a *compatibility* check for component-based design. When refining a design into an implementation, interface models require that the output behavior of a component satisfies the design specification only when the input assumptions of the specification are satisfied, yielding greater flexibility in the choice of implementations. Technically, our interface models are games between two players, Input and Output; the duality of the players accounts for the dual roles of inputs and outputs in composition and refinement. We present two interface models in detail, one for a simple synchronous form of interaction between components typical in hardware, and the other for more complex synchronous interactions on bidirectional connections. As an example, we specify the interface of a bidirectional bus, with the input assumption that at any time at most one compo-

nent has write access to the bus. For these interface models, we present algorithms for compatibility and refinement checking, and we describe efficient symbolic implementations.

2.1 Introduction

One of the main applications of modeling formalisms is to capture designs. We present *interface models* that are specifically geared to support the component-based approach to design. Interface models describe both the inputs that can be accepted by a component, and the outputs it can generate. As an interface constrains the acceptable inputs, the underlying component fits into some design contexts (which meet the constraints), but not into others. Interface models provide a means for answering four questions that arise in component-based design: the *well-formedness question* (can a component be used in some design, i.e., are the input constraints satisfiable?), the *verification question* (does a component satisfy a given property in all designs?), the *compatibility question* (do two components interact in compatible ways in a design?), and the *refinement question* (can a component be substituted for another one in every design context without violating compatibility?).

For each of the questions of well-formedness, verification, compatibility, and refinement, there are two basic choices for treating inputs and outputs. The *graph* view quantifies inputs and outputs with the same polarity; the *game* view quantifies inputs and outputs with opposite polarities. In the graph view, both inputs and outputs can be seen as labels in a nondeterministic state transition graph; in the game view, inputs and outputs are chosen by different players and the result of each combination of choices determines the state transition. For example, the graph view is appropri-

ate for the verification question: does a component satisfy a given property for all acceptable inputs and all possible outputs? On the other hand, the game view is necessary for the well-formedness question [5, 61]: are there acceptable inputs for all possible choices of outputs? We argue that also for compatibility and refinement, the game view is the appropriate one.

2.1.1 The graph view

The graph view is taken by many process algebras (e.g., [94, 117]) and state-based models (e.g., [111, 109, 45, 17]). These frameworks are aimed at verification. Indeed, also refinement is typically viewed as a verification question: does a more detailed description of a component generate only behaviors that are permitted by a more abstract description? Refinement is usually defined as a form of trace containment or simulation: when quantifying universally over both inputs and outputs, we say that a component N refines a component M (written $N \preceq M$) if, for all input and output choices, the behaviors of N are a subset of those of M. In particular, N can only produce outputs that are also produced by M, and N can only accept inputs that are also accepted by M. This ensures that every language-theoretic property (such as safety) that holds for M also holds for N. The graph view of refinement, however, becomes problematic when we interpret refinement as substitutivity. The output clause is still appropriate: by requiring that the output behavior of N is a subset of that of M, it ensures that if the outputs of M can be accepted by the other components of the design, so can those of N. The input clause instead is questionable: it states that the implementation N should be able to accept a *subset* of the inputs accepted by the specification M. This raises the possibility that, when N is substituted for M in a design, N cannot accept some inputs from other components that could be

accepted by M. Hence, *substitutivity of refinement* does not hold in the graph view. Indeed, in process algebras and the modeling language SMV [45], if $N \preceq M$ and $M \| P$ is deadlock-free, it is possible that $N \| P$ deadlocks [9]. To remedy this situation, some models, such as *I/O automata* [110] and *reactive modules* [17], require that components be able to accept *all* possible inputs; this condition is known as *input-enabledness* or *receptivity*. This requirement forces models to specify the outputs generated in response to *all* possible inputs, including inputs that the designers know cannot occur in the actual design. In turn, this creates further complications in the study of refinement. It is natural to require that the output behavior of N is a subset of that of M, when these output behaviors occur in response to inputs that are possible in the design. Extending the requirement also to inputs that are known not to occur is instead problematic: such outputs are often chosen arbitrarily, with the goal of simplifying the models or their implementations (as in the case of hardware optimizations that exploit "don't care" information about inputs). Thus, in input-enabled or receptive approaches, refinement is generally checked only between *closed* systems, that have no input from the environment; open systems are first closed by composing them with specifically-designed components that represent the environment.

The graph view is also limited in its capability to analyze component compatibility. If models specify explicitly which inputs can be accepted, and which ones are *illegal*, then it is possible to ask the compatibility question generically: do illegal inputs occur? If we quantify universally over both inputs and outputs, we obtain a verification question: two components M and N are compatible if, once composed, they accept all inputs. This is not a natural phrasing of the compatibility question: it requires $M \| N$ to accept all inputs, even though M and N could have illegal in-

puts. A more compositional definition is to call M and N compatible if there are *some* input sequences that ensure that all illegal inputs of M and N are avoided, and to label all other sequences as illegal for $M\|N$. This definition of compatibility leads to a dual treatment of inputs (quantified existentially) and outputs (quantified universally), and to the game view.

2.1.2 The game view

According to the game view, inputs and outputs play dual roles. In trace theory [61], a trace model consists in two sets, of accepted and rejected traces, and games are used to solve the realizability and compatibility questions. In the game semantics of [8, 9] and the interface models of [54, 55], components are explicitly modeled as games between two players, Input and Output. The moves of Input represent the inputs that can be accepted, and the moves of Output the outputs that can be generated. To model the fact that these sets can change in time, after the input and output moves are chosen, the game moves to a new state, with possibly different sets of accepted inputs and possible outputs.

In the study of compatibility, game-based approaches quantify inputs existentially, and outputs universally. When two components M and N are composed, their composition may have illegal states, where one component emits outputs that are illegal inputs for the other one. Yet, M and N are considered *compatible* as long as there is *some* input behavior that ensures that, for all output behaviors, the illegal states are avoided: in other words, M and N are compatible if there is some environment in which they can be used correctly together. In turn, the input behaviors that ensure compatibility constitute the legal behaviors for the composition $M\|N$: when com-

posing component models, both the possible output behaviors, and the legal input behaviors, are composed.

The game view leads to an *alternating* view of refinement [19]: a more detailed component N refines an abstract component M if all legal inputs of M are also legal for N and if, when M and N are subject to the same (legal) inputs, N generates output behaviors that are a subset of those of M. This definition ensures that, whenever $N \preceq M$, we can substitute N for M in every design without creating any incompatibility: in the game view, substitutivity of refinement holds. The alternating definition of refinement also mirrors the contravariant definition of subtyping in programming languages, which also supports substitutivity [118]. Indeed, the game framework can be viewed as a generalization of type theory to behaviors.

2.1.3 Synchronous interface models

In this chapter, we adopt the game view to modeling, and we introduce two interface models for *synchronous* components. We begin with the simple model of *Moore interfaces*: in addition to the usual transition relation of a synchronous system, which describes the update rules for the outputs, a Moore interface has a symmetrically-defined transition relation for the inputs, which specifies which input transitions are acceptable. Our second model, *bidirectional interfaces*, illustrate how game-based models can be richer than their graph-based counterparts. Bidirectional connections cannot be modeled in the input-enabled setting: there are always environments that use such connections as input, and environments that use them as output, so that no component can work in all environments. Bidirectional connections, however, can be naturally modeled as a game between Input and Output players. As an example, we encode the access protocol to the PCI bus, in which several components share access

to a multi-directional bus. By checking the compatibility of the component models, we can ensure that no conflicts for bus access arise. We have implemented tools for symbolic compatibility and refinement checking for both Moore and bidirectional interfaces, and we discuss how the game-based algorithms can be implemented with minor modifications to the usual symbolic machinery for graph-based algorithms, and yield a similar efficiency.

Finally, interfaces enable us to encode the environment assumptions that are used in assume-guarantee reasoning directly as input assumptions of the interface, rather than as separate "environment" components. This leads to a verification rule for compositional refinement that combines, and generalizes, rules proposed in [17, 16]. The rule illustrates how the essence of compositional verification consists in studying *both* the implementation and the specification as constrained by their actual environments.

2.2 Compatibility and Composition

2.2.1 Moore interfaces

Moore interfaces model both the behavior of a system component, and the interface between the component and its environment. The state of a module is described by a set of *state variables,* partitioned into sets of *input* and *output* variables. The input variables represent inputs to the module, and their value can be read, but not changed, by the module; the output variables represent outputs of the module, and their value can be changed (and read) by the module. The possible changes of output variables are described by an *output transition relation,* while the legal changes of input variables are described by an *input transition relation.* Hence, the output

```
interface Counter
  output q0, q1: bool;
  input  cl:     bool;

  input atom
    init
      [] true -> cl :=nondet
    update
      [] true -> cl':=nondet
  endatom
  output atom
    init
      [] true -> q0:=1; q1:=1;
    update
      []  cl             -> q1':=1; q0':=1
      [] ~cl &  q1 &  q0 -> q1':=1; q0':=0
      [] ~cl &  q1 & ~q0 -> q1':=0; q0':=1
      [] ~cl & ~q1 &  q0 -> q1':=0; q0':=0
      [] ~cl & ~q1 & ~q0 -> q1':=1; q0':=1
  endatom
end interface
```

Figure 2.1. A counter modeled as a Moore interface. The guarded-command syntax is derived from the one of *reactive modules* [17] and Mocha [20, 53]; input atoms describe the input assumptions, and the output atoms describe the output behavior. When more than one guard is true, the command is selected nondeterministically. Input variables not mentioned by the command are updated nondeterministically.

```
interface Adder
  input  q0, q1: bool; di: [0..7];
  output do: [0..7];

  input atom
    init
      [] true -> q0:=1
      [] true -> q1:=1
    update
      [] true -> q0':=1
      [] true -> q1':=1
  endatom
  output atom
    init
      [] true -> do:=nondet
    update
      []  q0 &  q1 -> do':=di'
      [] ~q0 &  q1 -> do':=di'+1
      []  q0 & ~q1 -> do':=di'-1
  endatom
end interface
```

Figure 2.2. A ± 1 adder modeled as a Moore interface.

transition relation describes the module's behavior, and the input transition relation describes the input assumptions of the interface. Finally, a set of *initial outputs* specifies the initial condition of the module, and a set of *initial inputs* specifies the desired initial condition of the environment.

Example 2.1 We illustrate the features of Moore interfaces by modeling a simple example: a ± 1 adder driven by a binary counter. (Figures 2.1 and 2.2). The adder *Adder* has two control inputs q_0 and q_1, data inputs $i_7 \cdots i_0$, and data outputs $o_7 \cdots o_0$. When $q_0 = q_1 = 1$, the adder leaves the input unchanged: the next value of $o_7 \cdots o_0$ is equal to $i_7 \cdots i_0$. When $q_0 = 0$ and $q_1 = 1$, the next outputs are given by $[o'_7 \cdots o'_0] = [i_7 \cdots i_0] + 1 \bmod 2^8$, where primed variables denote the values at the next clock cycle, and $[o'_7 \cdots o'_0]$ is the integer encoded in binary by $o'_7 \cdots o'_0$. Similarly, when $q_1 = 0$ and $q_0 = 1$, we have $[o'_7 \cdots o'_0] = [i_7 \cdots i_0] - 1 \bmod 2^8$. The adder is designed with the assumption that q_1 and q_0 are not both 0: hence, the input transition relation of *Adder* states that $q'_0 q'_1 \neq 00$. In order to cycle between adding $0, +1, -1$, the control inputs q_0 and q_1 are connected to the outputs q_1 and q_0 of a two-bit count-to-zero counter *Counter*. The counter has only one input, *cl*: when $cl = 0$, then $q'_1 q'_0 = 11$; otherwise, $[q'_1 q'_0] = [q_1 q_0] - 1 \bmod 4$.

When the counter is connected to the adder, the joint system can take a transition to a state where $q_1 q_0 = 00$, violating the adder's input assumptions. In spite of this, the counter and the adder are compatible, since there is a way to use them together: to avoid the incompatible transition, it suffices to assert $cl = 0$ early enough in the count-to-zero cycle of the counter. To reflect this, when we compose *Counter* and *Adder*, we synthesize for their composition $Counter \| Adder$ a new input assumption, that ensures that the input assumptions of both *Counter* and *Adder* are satisfied. To determine the new input assumption, we solve a game between Input, which chooses

the next values of cl and $i_7 \cdots i_0$, and Output, which chooses the next values of q_0, q_1, and $o_7 \cdots o_0$. The goal of Input is to avoid a transition to $q_1 q_0 = 00$. At the states where $q_1 q_0 = 01$, Input can win if $cl = 0$, since at the next clock cycle we will have $q_1' q_0' = 11$; but Input cannot win if $cl = 1$. By choosing $cl' = 0$, Input can also win from the states where $q_1 q_0 = 10$. Finally, Input can always win from the states where $q_1 q_0 = 11$, for all cl'. Thus, we associate with $Counter \| Adder$ a new input assumption encoded by the transition relation requiring that whenever $q_1 q_0 = 10$, then $cl' = 0$. The input requirement $q_1 q_0 \neq 00$ of the adder gives rise, in the composite system, to the requirement that the reset-to-1 occurs early in the count-to-zero cycle of the counter. ∎

Given a set $\mathcal{W}$ of typed variables with finite domain, a state s over $\mathcal{W}$ is a function that assigns to each $x \in \mathcal{W}$ a value $s[\![x]\!]$ of the appropriate type; we write $\mathcal{S}[\mathcal{W}]$ for the set of all states over $\mathcal{W}$. We denote by $\mathcal{W}' = \{x' \mid x \in \mathcal{W}\}$ the set obtained by priming each variable in $\mathcal{W}$; given a predicate φ on $\mathcal{W}$, we denote by φ' the predicate on $\mathcal{W}'$ obtained by replacing in φ every $x \in \mathcal{W}$ with $x' \in \mathcal{W}'$. Given a state $s \in \mathcal{S}[\mathcal{W}]$ and a predicate φ on $\mathcal{W}$, we write $s \models \varphi$ if φ is satisfied under the variable interpretation specified by s. Given two states $s, s' \in \mathcal{S}[\mathcal{W}]$ and a predicate φ on $\mathcal{W} \cup \mathcal{W}'$, we write $(s, s') \models \varphi$ if φ is satisfied by the interpretation that assigns to $x \in \mathcal{W}$ the value $s[\![x]\!]$, and to $x' \in \mathcal{W}'$ the value $s'[\![x]\!]$. Moore interfaces are defined as follows.

Definition 2.1 (Moore interface) A *Moore interface* $M = \langle \mathcal{V}_M^i, \mathcal{V}_M^o, \theta_M^i, \theta_M^o, \tau_M^i, \tau_M^o \rangle$ consists of the following components:

- A finite set $\mathcal{V}_M^i$ of *input variables,* and a finite set $\mathcal{V}_M^o$ of *output variables.* The two sets must be disjoint; we define $\mathcal{V}_M = \mathcal{V}_M^i \cup \mathcal{V}_M^o$.

- A satisfiable predicate θ^i_M on $\mathcal{V}^i_M$ defining the legal initial values for the input variables, and a satisfiable predicate θ^o_M on $\mathcal{V}^o_M$ defining the initial values for the output variables.

- An *input transition predicate* τ^i_M on $\mathcal{V}_M \cup (\mathcal{V}^i_M)'$, specifying the legal updates for the input variables, and an *output transition predicate* τ^o_M on $\mathcal{V}_M \cup (\mathcal{V}^o_M)'$, specifying how the module can update the values of the output variables. We require that the formulas $\forall \mathcal{V}_M . \exists (\mathcal{V}^i_M)' . \tau^i_M$ and $\forall \mathcal{V}_M . \exists (\mathcal{V}^o_M)' . \tau^o_M$ hold. ∎

The above interfaces are called *Moore* because the next value of the output variables can depend on the current state, but not on the next value of the input variables, as in Moore machines. The requirements on the input and output transition relations ensure that the interface is non-blocking: from every state there is some legal input and possible output. Given a Moore interface $M = \langle \mathcal{V}^i_M, \mathcal{V}^o_M, \theta^i_M, \theta^o_M, \tau^i_M, \tau^o_M \rangle$, we let $\mathit{Traces}(\mathcal{V}^i_M, \mathcal{V}^o_M, \theta^i_M, \theta^o_M, \tau^i_M, \tau^o_M)$ be the set of *traces* of M, consisting of all the infinite sequences $s_0, s_1, s_2, \ldots$ of states of $\mathcal{S}[\mathcal{V}_M]$ such that $s_0 \models \theta^i_M \wedge \theta^o_M$, and $(s_k, s_{k+1}) \models \tau^i_M \wedge \tau^o_M$ for all $k \geq 0$.

Composition of Moore interfaces. Two Moore interfaces M and N are *composable* if $\mathcal{V}^o_M \cap \mathcal{V}^o_N = \emptyset$. If M and N are composable, we merge them into a single interface P as follows. We let $\mathcal{V}^o_P = \mathcal{V}^o_M \cup \mathcal{V}^o_N$ and $\mathcal{V}^i_P = (\mathcal{V}^i_M \cup \mathcal{V}^i_N) \setminus \mathcal{V}^o_P$. The output behavior of P is simply the joint output behavior of M and N, since each interface is free to choose how to update its output variables: hence, $\theta^o_P = \theta^o_M \wedge \theta^o_N$ and $\tau^o_P = \tau^o_M \wedge \tau^o_N$. On the other hand, we cannot simply adopt the symmetrical definition for the input assumptions. A syntactic reason is that $\theta^i_M \wedge \theta^i_N$ and $\tau^i_M \wedge \tau^i_N$ may contain variables in $(\mathcal{V}^o_P)'$. But a deeper reason is that we may need to strengthen the input assumptions of P further, in order to ensure that the input assumptions

of M and N hold. If we can find such a further strengthening θ^i and τ^i, then M and N are said to be *compatible,* and $P = M\|N$ with θ^i_P and τ^i_P being the weakest such strengthenings; otherwise, we say that M and N are incompatible, and $M\|N$ is undefined. Hence, informally, M and N are compatible if they can be used together under some assumptions.

Definition 2.2 (Compatibility and composition of Moore interfaces) For any two Moore interfaces M and N, we say that M and N are *composable* if $\mathcal{V}^o_M \cap \mathcal{V}^o_N = \emptyset$. If M and N are composable, let $\mathcal{V}^o_P = \mathcal{V}^o_M \cup \mathcal{V}^o_N$, $\mathcal{V}^i_P = (\mathcal{V}^i_M \cup \mathcal{V}^i_N) \setminus \mathcal{V}^o_P$, $\mathcal{V}_P = \mathcal{V}^o_P \cup \mathcal{V}^i_P$, $\theta^o_P = \theta^o_M \wedge \theta^o_N$, and $\tau^o_P = \tau^o_M \wedge \tau^o_N$.

The interfaces M and N are *compatible* (written $M \wr\wr N$) if they are composable, and if there are predicates θ^i on $\mathcal{V}^i_P$ and τ^i on $\mathcal{V}_P \cup (\mathcal{V}^i_P)'$ such that (i) θ^i is satisfiable; (ii) $\forall \mathcal{V}_P . \exists (\mathcal{V}^i_P)' . \tau^i$ holds; (iii) for all $s_0, s_1, s_2, \ldots \in \mathit{Traces}(\mathcal{V}^i_P, \mathcal{V}^o_P, \theta^i, \theta^o_P, \tau^i, \tau^o_P)$ we have $s_0 \models \theta^i_M \wedge \theta^i_N$ and, for all $k \geq 0$, $(s_k, s_{k+1}) \models \tau^i_M \wedge \tau^i_N$.

The *composition* $P = M\|N$ is defined if and only if $M \wr\wr N$, in which case P is obtained by taking for the input predicate θ^i_P and for the input transition relation τ^i_P the weakest predicates such that the above condition holds. ∎

To compute $M\|N$, we consider a game between Input and Output. At each round of the game, Output chooses new values for the output variables $\mathcal{V}^o_P$ according to τ^o_P; simultaneously and independently, Input chooses (unconstrained) new values for the input variables $\mathcal{V}^i_P$. The goal of Input is to ensure that the resulting behavior satisfies $\theta^i_M \wedge \theta^i_P$ at the initial state, and $\tau^i_M \wedge \tau^i_N$ at all state transitions. If Input can win the game, then M and N are compatible, and the most general strategy for Input will give rise to θ^i_P and τ^i_P; otherwise, M and N are incompatible. The algorithm for computing θ^i_P and τ^i_P proceeds by computing iterative approximations to τ^i_P, and

to the set C of states from which Input can win the game. We let $C_0 = \mathrm{T}$ and, for $k \geq 0$:

$$\widetilde{\tau}_{k+1} = \forall(\mathcal{V}_P^o)'.\Big(\tau_P^o \rightarrow (\tau_M^i \wedge \tau_N^i \wedge C_k')\Big) \qquad C_{k+1} = C_k \wedge \exists(\mathcal{V}_P^i)'.\widetilde{\tau}_{k+1}. \tag{2.1}$$

Note that $\widetilde{\tau}_{k+1}$ is a predicate on $\mathcal{V}_P^o \cup \mathcal{V}_P^i \cup (\mathcal{V}_P^i)'$. Hence, $\widetilde{\tau}_{k+1}$ ensures that, regardless of how $\mathcal{V}_P^o$ are chosen, from C_{k+1} we have that (i) for one step, τ_M^i and τ_N^i are satisfied; and (ii) the step leads to C_k. Thus, indicating by $C_* = \lim_{k\to\infty} C_k$ and $\widetilde{\tau}_* = \lim_{k\to\infty} \widetilde{\tau}_k$ the fixpoints of (2.1) we have that C_* represents the set of states from which Input can win the game, and $\widetilde{\tau}_*$ represents the most liberal Input strategy for winning the game. This suggests us to take $\tau_P^i = \widetilde{\tau}_*$. However, this is not always the weakest choice, as required by Definition 2.2: a weaker choice is $\tau_P^i = \neg C_* \vee \widetilde{\tau}_*$, or equivalently $\tau_P^i = C_* \rightarrow \widetilde{\tau}_*$. Contrary to $\tau_P^i = \widetilde{\tau}_*$, this weaker choice ensures that the interface P is non-blocking. We remark that the choices $\tau_P^i = \widetilde{\tau}_*$ and $\tau_P^i = C_* \rightarrow \widetilde{\tau}_*$ differ only at non-reachable states. Since the state-space of P is finite, by monotonicity of (2.1) we can compute the fixpoint C_* and $\widetilde{\tau}_*$ in a finite number of iterations. Finally, we define the input initial condition of P by $\theta_P^i = \forall\mathcal{V}^o.(\theta_P^o \rightarrow (\theta_M^i \wedge \theta_N^i \wedge C_*))$. The following algorithm summarizes these results.

Algorithm 2.1 Given two composable Moore interfaces M and N, let $C_0 = \mathrm{T}$, and for $k > 0$, let the predicates C_k and $\widetilde{\tau}_k$ be as defined by (2.1). Let $\widetilde{\tau}_* = \lim_{k\to\infty} \widetilde{\tau}_k$ and $C_* = \lim_{k\to\infty} C_k$; the limits can be computed with a finite number of iterations, and let $\theta_*^i = \forall\mathcal{V}^o.\Big(\theta_P^o \rightarrow (\theta_M^i \wedge \theta_N^i \wedge C_*)\Big)$. Then the interfaces M and N are *compatible* iff θ_*^i is satisfiable; in this case their composition $P = M\|N$ is given by

$$\begin{array}{llll} \mathcal{V}_P^o &= \mathcal{V}_M^o \cup \mathcal{V}_N^o & \tau_P^o &= \tau_M^o \wedge \tau_N^o \qquad \theta_P^o = \theta_M^o \wedge \theta_N^o \\ \mathcal{V}_P^i &= (\mathcal{V}_M^i \cup \mathcal{V}_N^i) \setminus \mathcal{V}^o & \tau_P^i &= C_* \rightarrow \widetilde{\tau}_* \qquad \theta_P^i = \theta_*^i. \end{array}$$ ∎

Implementation considerations. We have implemented composition and compatibility checking for Moore interfaces by extending the Mocha model checker [20] to interfaces. To obtain an efficient implementation, we represent both the input and the output transition relations using a conjunctively decomposed representation, where a relation τ is represented by a list of BDDs $\tau_1, \tau_2, \ldots, \tau_n$ such that $\tau = \wedge_{i=1}^{n} \tau_i$. When computing $P = M \| N$, the list for τ_P^o can be readily obtained by concatenating the lists for τ_M^o and τ_N^o. Moreover, assume that τ_P^o is represented as $\bigwedge_{i=1}^{n} \tau_i^o$, and that $\tau_M^i \wedge \tau_N^i$ is represented as $\bigwedge_{j=1}^{m} \tau_j^o$. Given C_k, from (2.1) we obtain the conjunctive decomposition $\bigwedge_{j=1}^{m+1} \widetilde{\tau}_{k+1,j}$ for $\widetilde{\tau}_{k+1}$ by taking $\widetilde{\tau}_{k+1,m+1} = \neg\exists(\mathcal{V}_P^o)'.(\tau_P^o \wedge \neg C_k')$ and, for $1 \leq j \leq m$, by taking $\widetilde{\tau}_{k+1,j} = \neg\exists(\mathcal{V}_P^o)'.(\tau_P^o \wedge \neg\tau_j^i)$. We also obtain $C_{k+1} = \exists(\mathcal{V}_P^i)'.\bigwedge_{j=1}^{m+1} \widetilde{\tau}_{k+1,j}$. All these operations can be performed using image computation techniques. Once we reach k such that $C_k \equiv C_{k+1}$, the BDDs $\widetilde{\tau}_{k,1}, \ldots, \widetilde{\tau}_{k,m+1}$ form a conjunctive decomposition for $\widetilde{\tau}_*$. Since the two transition relations $\widetilde{\tau}_*$ and $C_* \rightarrow \widetilde{\tau}_*$ differ only for the behavior at non-reachable states, in our implementation we take directly $\tau_P^i = \widetilde{\tau}_*$, obtaining again a conjunctive decomposition. With these techniques, the size (number of BDD variables) of the interfaces that our tool is able to check for compatibility, and compose, is roughly equivalent to the size of the models that Mocha [20] can verify with respect to safety properties.

2.2.2 Bidirectional Interfaces

Bidirectional interfaces model components that have bidirectional connections. To model bidirectionality we find it convenient to add to the Moore model a set Q of *locations*. Informally, each location $q \in Q$ partitions the interface variables into inputs and outputs, and determines what values are legal for the inputs, and what values can be assigned to the outputs. At each location $q \in Q$, a particular choice of

output and input values determines the successor location q'. The precise definition is as follows.

Definition 2.3 (Bidirectional interfaces) A *bidirectional interface* M is a tuple $\langle \mathcal{V}_M, Q_M, \hat{q}_M, v^o_M, \phi^i_M, \phi^o_M, \rho_M \rangle$ consisting of the following components:

- A finite set $\mathcal{V}_M$ of input or output *(inout)* variables.
- A finite set Q_M of locations, including an initial location $\hat{q}_M \in Q_M$.
- A function $v^o_M : Q_M \to 2^{\mathcal{V}_M}$, that associates with all $q \in Q_M$ the set $v^o_M(q)$ of variables that are used as outputs at location q. For all $q \in Q_M$, we denote by $v^i_M(q) = \mathcal{V}_M \setminus v^o_M(q)$ the set of variables that are used as inputs.
- Two labelings ϕ^i_M and ϕ^o_M, which associate with each location $q \in Q_M$ a predicate $\phi^i_M(q)$ on $v^i_M(q)$, called the input assumption, and a predicate $\phi^o_M(q)$ on $v^o_M(q)$, called the output guarantee. For all $q \in Q_M$, both $\phi^i_M(q)$ and $\phi^o_M(q)$ should be satisfiable.
- A labeling ρ_M, which associates with each pair of locations $q, r \in Q_M$ a predicate $\rho_M(q, r)$ on $\mathcal{V}_M$, called the *transition guard.* We require that for every location $q \in Q_M$, (i) the disjunction $\bigvee_{r \in Q_M} \rho_M(q, r)$ is valid and (ii) $\forall r, r' \in Q_M$, $(r \neq r') \Rightarrow \neg(\rho_M(q, r) \wedge \rho_M(q, r'))$. Condition (i) ensures that the interface is non-blocking, and condition (ii) ensures determinism. ∎

We let $\mathcal{V}^i_M = \bigcup_{q \in Q_M} v^i_M(q)$ and $\mathcal{V}^o_M = \bigcup_{q \in Q_M} v^o_M(q)$ be the sets of all variables that are ever used as inputs or outputs (note that we do not require $\mathcal{V}^i_M \cap \mathcal{V}^o_M = \emptyset$). We define the set $\mathit{Traces}(\langle \mathcal{V}_M, Q_M, \hat{q}_M, \phi^i_M, \phi^o_M, \rho_M \rangle)$ of *bidirectional traces* to be the set of infinite sequences $q_0, s_0, q_1, s_1, \ldots$, where $q_0 = \hat{q}_M$, and for all $k \geq 0$,

we have $q_k \in Q_M$, $s_k \in \mathcal{S}[\mathcal{V}_M]$, and $s_k \models (\phi^i_M(q_k) \wedge \phi^o_M(q_k) \wedge \rho_M(q_k, q_{k+1}))$. For $q_0, s_0, q_1, s_1, \ldots \in \mathit{Traces}(\langle \mathcal{V}_M, Q_M, \hat{q}_M, \phi^i_M, \phi^o_M, \rho_M \rangle)$ and $k \geq 0$, we say that q_k is *reachable* in $\langle \mathcal{V}_M, Q_M, \hat{q}_M, \phi^i_M, \phi^o_M, \rho_M \rangle$.

Composition of bidirectional interfaces is defined along the same lines as for Moore interfaces. Local incompatibilities arise not only when one interface output values do not satisfy the input assumptions of the other, but also when the same variable is used as output by both interfaces. The formal definition follows.

Definition 2.4 (Composition of bidirectional interfaces)
Given two bidirectional interfaces M and N, let $\mathcal{V}_\otimes = \mathcal{V}_M \cup \mathcal{V}_N$, $Q_\otimes = Q_M \times Q_N$, and $\hat{q}_\otimes = (\hat{q}_M, \hat{q}_N)$. For all $(p, q) \in Q_M \times Q_N$, let $\phi^o_\otimes(p, q) = \phi^o_M(p) \wedge \phi^o_M(q)$, and for all $(p', q') \in Q_M \times Q_N$, let $\rho_\otimes((p, q), (p', q')) = \rho_M(p, p') \wedge \rho_N(p, p')$. The interfaces M and N are *compatible* (written $M \between N$) if there is a labeling ψ associating with all $(p, q) \in Q_\otimes$ a predicate $\psi(p, q)$ on $\mathcal{V}_\otimes \setminus (\mathcal{V}^o_M(p) \cup \mathcal{V}^o_N(q))$ such that (i) $\psi(p, q)$ is satisfiable at all $(p, q) \in Q_\otimes$, and (ii) all traces $(p_0, q_0), s_0, (p_1, q_1), s_1, (p_2, q_2), s_2, \ldots \in \mathit{Traces}(\mathcal{V}_\otimes, Q_\otimes, \hat{q}_\otimes, \psi, \phi^o_\otimes, \rho_\otimes)$ satisfy, for all $k \geq 0$, the conditions (a) $\mathcal{V}^o_M(p_k) \cap \mathcal{V}^o_N(q_k) = \emptyset$ and (b) $s_k \models \phi^i_M(p_k) \wedge \phi^i_N(q_k)$. The composition $P = M \| N$ is defined if and only if M and N are compatible; if they are, then $P = M \| N$ is obtained by taking for ϕ^i_P the weakest predicate ψ such that the above conditions (a) and (b) on traces hold, by taking for Q_P the subset of locations of $Q_\otimes$ that are reachable in $\langle \mathcal{V}_\otimes, Q_\otimes, \hat{q}_\otimes, \mathcal{V}^o_\otimes, \phi^i_P, \phi^o_\otimes, \rho_\otimes \rangle$, by taking $\mathcal{V}_P = \mathcal{V}_\otimes$ and $\hat{q}_P = \hat{q}_\otimes$, and by taking for $\mathcal{V}^o_P, \phi^i_P, \phi^o_P$, and ρ_P the restrictions of $\mathcal{V}^o_\otimes, \phi^i_\otimes, \phi^o_\otimes$, and $\rho_\otimes$ to Q_P. ∎

Algorithm 2.2 Given two bidirectional interfaces M and N, let $\mathcal{V}_\otimes = \mathcal{V}_M \cup \mathcal{V}_N$, $Q_\otimes = Q_M \times Q_N$, and $\hat{q}_\otimes = (\hat{q}_M, \hat{q}_N)$. For all $(p, q) \in Q_M \times Q_N$, let $\phi^i_\otimes(p, q) = \phi^i_M(p) \wedge \phi^i_M(q)$, and for all $(p', q') \in Q_M \times Q_N$, let $\rho_\otimes((p, q), (p', q')) = \rho_M(p, p') \wedge \rho_N(p, p')$. The

input labeling $\phi^i_\otimes(p,q)$ is computed by repeating the following steps, that progressively strengthen the input assertions:

> [Step 1] For all $(p,q) \in Q_M \times Q_N$, if $v^o_M(p) \cap v^o_N(q) \neq \emptyset$, then initialize $\phi^i_\otimes(p,q)$ to F; otherwise initialize $\phi^i_\otimes(p,q)$ to the predicate $\forall v^o_\otimes(p,q).(\phi^o_\otimes(p,q) \rightarrow (\phi^i_M(p) \wedge \phi^i_N(q))$.
>
> [Step 2] For all (p,q) and (p',q') in $Q_M \times Q_N$, if $\phi^i_\otimes(p',q')$ is unsatisfiable, then replace $\phi^i_\otimes(p,q)$ with $\phi^i_\otimes(p,q) \wedge \forall v^o_\otimes(p,q).(\phi^o_\otimes(p,q) \rightarrow \neg\rho_\otimes((p,q),(p',q'))$.
>
> Repeat [Step 2] until all input assumptions are replaced by equivalent predicates, i.e., are not strengthened.

We have that $M \wr\wr N$ iff $\phi^i_\otimes(\hat{q}_M, \hat{q}_N)$ is satisfiable. If $M \wr\wr N$ then their composition P is defined by taking Q_P to be the subset of locations of $Q_\otimes$ that are reachable in $\langle \mathcal{V}_\otimes, Q_\otimes, \hat{q}_\otimes, v^o_\otimes, \phi^i_P, \phi^o_\otimes, \rho_\otimes \rangle$, by taking $\mathcal{V}_P = \mathcal{V}_\otimes$ and $\hat{q}_P = \hat{q}_\otimes$, and by taking for $v^o_P, \phi^i_P, \phi^o_P$, and ρ_P the restrictions of $v^o_\otimes, \phi^i_\otimes, \phi^o_\otimes$, and $\rho_\otimes$ to Q_P. ∎

We have developed and implemented symbolic algorithms for composition and compatibility and refinement checking of bidirectional interfaces. The tool, written in Java, is based on the CUDD Package used in JMocha [53]. In our implementation, the locations are represented explicitly, while the input assumptions and output guarantees at each location are represented and manipulated symbolically. This hybrid representation is well-suited to the modeling of bidirectional interfaces, where the set of input and output variables depends on the location.

Example 2.2 (PCI Bus) We consider a PCI bus configuration with two PCI-compliant master devices and a PCI arbiter as shown in Figure 2.3(a). Each PCI master device has an *gnt* input and a *req* output to communicate with the arbiter, and a set of shared (read-write) signals, the `IRDY` and the `FRAME`, which are used

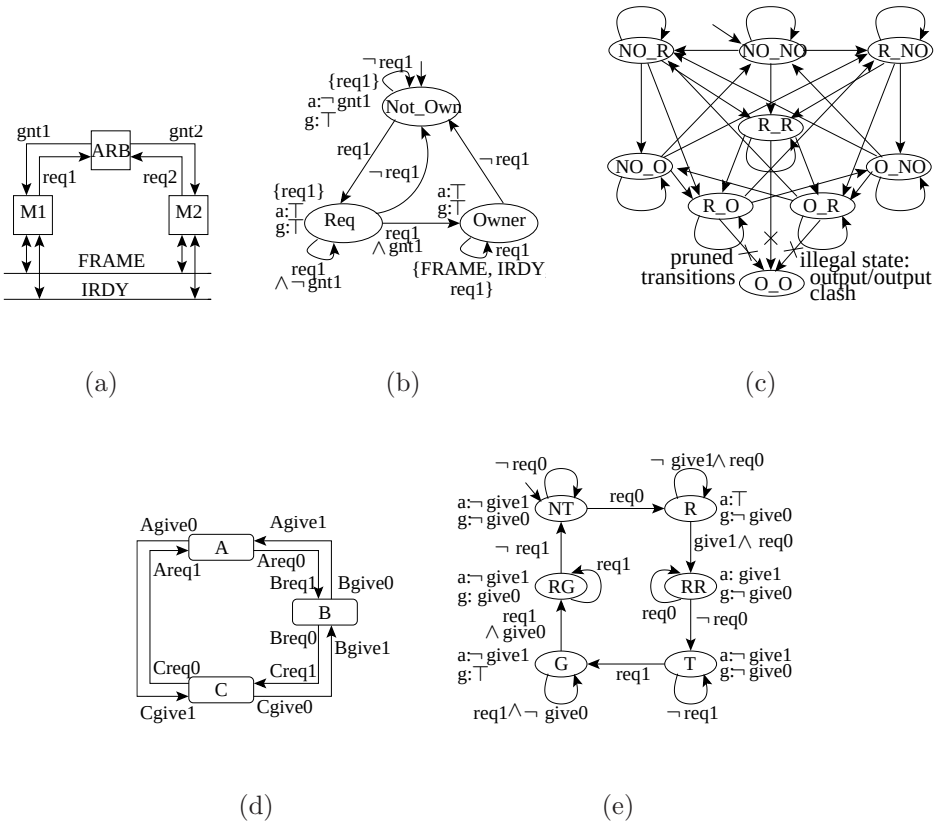


Figure 2.3. PCI and Token-ring Protocols 2.3(a) PCI Local Bus Structural Diagram 2.3(b) PCI Master Interface 2.3(c) Composite interface for two PCI Master Modules 2.3(d) Token Ring Network Configuration 2.3(e) Token-ring NT Interface

to communicate with target devices. The arbiter ensures that at most one master device can write to the shared signals. Figure 2.3(b) shows a graphical description of the interface representing a master device. The figure shows for each location, the assumption ("a"), the guarantee ("g"), the set of inout variables that the interface writes to, and guarded transitions between locations. Composing two such interfaces we obtain the interface shown in Figure 2.3(c). Location `Owner_Owner` is illegal because both components write the shared variables `FRAME` and `IRDY`. Input assumptions of locations `Req_Req`, `Owner_Req` and `Req_Owner` are strengthened to make the illegal location unreachable. Note that this propagates the PCI master's

assumptions about its environment to an assumption on the behavior of the arbiter (which is the environment of the composite module): the arbiter should never assert `gnt1` (`gnt2`) during or after asserting `gnt2` (`gnt1`), until `req2` (`req1`) is de-asserted at least once. ∎

2.2.3 Properties of compatibility and composition

If M and N are composable Moore interfaces, we define their *product* $M \otimes N$ by $\mathcal{V}^o_{M\otimes N} = \mathcal{V}^o_M \cup \mathcal{V}^o_N$ and $\mathcal{V}^i_{M\otimes N} = (\mathcal{V}^i_M \cup \mathcal{V}^i_N) \setminus \mathcal{V}^o_{M\otimes N}$, and by letting $\theta^o_{M\otimes N} = \theta^o_M \wedge \theta^o_N$, $\theta^i_{M\otimes N} = \theta^i_M \wedge \theta^i_N$, $\tau^o_{M\otimes N} = \tau^o_M \wedge \tau^o_N$, and $\tau^i_{M\otimes N} = \tau^o_M \wedge \tau^i_N$. Intuitively, an *environment* for a Moore interface M is an interface that drives all free inputs of M, ensuring that all the input assumptions are met. Precisely, we say that a Moore interface N is an environment for a Moore interface M if M and N are composable and closed (i.e., $\mathcal{V}^o_M \cap \mathcal{V}^o_N = \emptyset$, and $\mathcal{V}^i_{M\otimes N} = \emptyset$), and if the following conditions hold:

- *Non-blocking:* $\theta^i_{M\otimes N}$ is satisfiable, and $\forall \mathcal{V}^o_{M\otimes N}.(\exists \mathcal{V}^i_{M\otimes N})'.\tau^i_{M\otimes N}$ holds.
- *Legal:* for all sequences $s_0, s_1, s_2, \ldots$ of states in $\mathcal{S}[\mathcal{V}_{M\otimes N}]$ with $s_0 \models \theta^o_{M\otimes N}$ and $(s_k, s_{k+1}) \models \tau^o_{M\otimes N}$ for all $k \geq 0$, we have also that $s_0 \models \theta^i_{M\otimes N}$ and $(s_k, s_{k+1}) \models \tau^i_{M\otimes N}$ for all $k \geq 0$.

Analogous definitions for product and environment can be given for bidirectional interfaces. The following theorem states the main properties of compatibility and composition of Moore interfaces; an analogous result holds for bidirectional interfaces.

Theorem 2.1 (Properties of compatibility and composition) *The following assertions hold:*

1. Associativity: *Given three Moore interfaces M, N, P, either $(M\|N)\|P$ and $(M\|N)\|P$ are both undefined (due to non-composability or incompatibility), or they are both defined, in which case they are equal.*

2. Compatibility as existence of environment: *Given two composable Moore interfaces M and N, we have that $M \wr\!\wr N$ iff there is an environment for $M \otimes N$.*

3. Composition and input assumptions: *Given two compatible Moore interfaces M and N, and P composable with $M\|N$, we have that $(M\|N) \wr\!\wr P$ iff there is an environment for $M \otimes N \otimes P$.*

The second assertion makes precise our statement that two interfaces are compatible iff there is some environment in which they can work correctly together. The third assertion states that composition does not unduly restrict the input assumptions: checking compatibility with the composition $M\|N$ amounts to checking compatibility with M and N.

2.3 Refinement

The refinement relation aims at formalizing the relation between abstract and concrete versions of the same component, or between an abstract component and its implementation. In the input-enabled (or pessimistic) setting, refinement is usually defined as trace containment or simulation [116]: this ensures that the set of output behaviors of the implementation is a subset of that of the abstract component. However, such definitions are not be appropriate in a non-input-enabled setting such as our interfaces: they would also require that the set of legal inputs of the implementation is a subset of that of the abstract component — implying that the

implementation can be used in fewer environments than the abstract component was designed for. For example, if we adopted the classical definition, then the component *Adder* of Figure 2.1 would be refined by a component *AdderOnly* having the same output transition relation, but that does not perform subtraction: precisely, with the *single* input guard `[] true -> q1':=1`. As this example points out, refinement should be defined in a contravariant fashion: the implementation should accept more inputs, and produce fewer outputs, than the specification [54, 55].

We define refinement as alternating simulation [19]: roughly, a component N refines M (written $N \preceq M$) if N can simulate all inputs of M, and if M can simulate all outputs of N. Encoding the relation between the states of two Moore interfaces M and N by a predicate R, we can state the definition of refinement as follows.

Definition 2.5 (Refinement of Moore interfaces) Given two Moore interfaces M and N, we have that $N \preceq M$ if $\mathcal{V}_N^i \subseteq \mathcal{V}_M^i$ and $\mathcal{V}_M^i \cap (\mathcal{V}_M^o \cup \mathcal{V}_N^o) = \emptyset$, and if there is a predicate R on $\mathcal{V}_M \cup \mathcal{V}_N$ such that the following formulas are valid:

$$\begin{aligned} \theta_M^i \wedge \theta_N^o &\;\rightarrow\; \exists(\mathcal{V}_M^o \setminus \mathcal{V}_N^o).(\theta_N^i \wedge \theta_M^o \wedge R) \\ R \wedge \tau_M^i \wedge \tau_N^o &\;\rightarrow\; \exists(\mathcal{V}_M^o \setminus \mathcal{V}_N^o)'.(\tau_N^i \wedge \tau_M^o \wedge R') \end{aligned} \qquad \blacksquare$$

As for normal simulation, there is a unique largest refinement relation between any two Moore interfaces. Hence, Definition 2.5 provides an iterative algorithm for deciding refinement: let $R_0 = \mathrm{T}$, and for $k \geq 0$, let

$$R_{k+1} = R_k \wedge \forall(\mathcal{V}_M \cup \mathcal{V}_N)'.\Big(\tau_M^i \wedge \tau_N^o \rightarrow \exists(\mathcal{V}_M^o \setminus \mathcal{V}_N^o)'.(\tau_N^i \wedge \tau_M^o \wedge R_k')\Big). \qquad (2.2)$$

Denoting with $R_* = \lim_{k\to\infty} R_k$ the fixpoint (that again can be computed in a finite number of iterations), we have that $N \preceq M$ if and only if (i) $\mathcal{V}_N^i \subseteq \mathcal{V}_M^i$ and $\mathcal{V}_M^i \cap (\mathcal{V}_M^o \cup \mathcal{V}_N^o) = \emptyset$, and (ii) $\theta_M^i \wedge \theta_N^o \;\rightarrow\; \exists(\mathcal{V}_M^o \setminus \mathcal{V}_N^o).(\theta_N^i \wedge \theta_M^o \wedge R)$. In order to obtain

an efficient implementation, we can again take advantage for the computation of (2.2) of list representations for the transition relations, and apply image-computation techniques.

Refinement of bidirectional interfaces is defined similarly, except that the refinement relation relates the locations of the two interfaces, rather than the states. The definition is as follows.

Definition 2.6 (Refinement of bidirectional interfaces) Given two bidirectional interfaces M and N, N refines M ($N \preceq M$) iff there is a binary relation $\preceq \subseteq Q_N \times Q_M$ such that $\hat{q}_N \preceq \hat{q}_M$, and such that for all $q \preceq p$ we have (i) $v^i_N(q) \subseteq v^i_M(q)$, (ii) $v^o_N(q) \supseteq v^o_M(p)$, (iii) $\phi^i_M(p) \rightarrow \phi^i_N(q)$, (iv) $\phi^o_N(q) \rightarrow \phi^o_M(p)$, (v) for all $s \in \mathcal{S}[v^i_M(p)]$ and all $t \in \mathcal{S}[v^o_N(q)]$, if $s \models \rho_M(p, p')$ and $t \models \rho_N(q, q')$, then $q' \preceq p'$. ∎

We can check whether $N \preceq M$ by adapting the classical iterative refinement check [116]. We start with the total relation $\preceq_0 = Q_N \times Q_M$, and for $k \geq 0$, we let $\preceq_{k+1}$ be the subset of $\preceq_k$ such that conditions (i)–(v) hold, with $\preceq_k$ in place of $\preceq$ in condition (v). Once we reach $m \geq 0$ such that $\preceq_{m+1} = \preceq_m$, we have that $N \preceq M$ iff $\hat{q}_N \preceq \hat{p}_N$. Since bidirectional interfaces are deterministic we can reduce the refinement checking problem to graph reachability on the product interface and hence $N \preceq M$ can be decided in $O(|Q_N| \times |Q_M|)$ time.

Example 2.3 (Token Ring) The IEEE 802.5 (Token Ring) is a widely used deterministic LAN protocol. Figure 2.3(e) shows an interface modeling a node that initially does not have the token. The same diagram with T as initial state would represent a node that initially has the token. We call these two interfaces NT and

Algorithm 1 Refinement Check for Stateful Bidirectional Interfaces

Require: two stateful bidirectional interfaces $F = (P_F, Q_F, \hat{q}_F, o_F, o_F^+, \phi_F, \psi_F, \delta_F)$ and $F' = (P_{F'}, Q_{F'}, \hat{q}_{F'}, o_{F'}, o_{F'}^+, \phi_{F'}, \psi_{F'}, \delta_{F'})$

1: Let $Prod_{F,F'} = F \circledast F' =$ a *directed graph* $G(V_{Prod}, E_{Prod})$ where $V_{Prod} = Q_F \times Q_{F'}$ and $E_{Prod} \subseteq V_{Prod} \times V_{Prod}$ such that $e = (v_1, v_2) \in E_{Prod}$ iff $v_1 = (q_1, q_1')$, $v_2 = (q_2, q_2')$ and $\exists$ valuations $i \in [P_F - o_F^+(q_1)]$, $o' \in [o_{F'}(q_1']$ such that $\phi_F(q_1)@i$ and $\psi_{F'}(q_1')@o'$ and $q_2 = \delta_F(q_1, i \uplus o')$ and $q_2' = \delta_{F'}(q_1', i \uplus o')$.

2: Let a state $v = (q, q')$ be an *error* state iff $(o_{F'}(q') \not\supseteq o_F(q)) \vee (o_{F'}^+(q') \not\subseteq o_F^+(q)) \vee (P_{F'} - o_{F'}^+(q') \not\subseteq P_F - o_F^+(q)) \vee (\phi_F(q) \not\Rightarrow \phi_{F'}(q')) \vee (\psi_{F'}(q') \not\Rightarrow \psi_F(q))$.

3: **if** any error state is reachable from $(\hat{q}_F, \hat{q}_{F'})$ in $G(V_{Prod}, E_{Prod})$ **then**

4: F' does not refine F

5: **else**

6: F' refines F

T, respectively. The token ring components are connected in a cyclic network; each pair of adjacent nodes communicate by *req* and *gnt* signals (Figure 2.3(d)). The *req* signal flows clockwise, and is used to request the token; the signal *give* flows counterclockwise, and is used to grant the token. The protocol fails if more than one node has the token simultaneously: indeed, we can verify that two T interfaces are not compatible, while an NT interface is compatible with a T interface. Moreover, the protocol works for any number of participating nodes. To verify this, we check two refinements: first, an open-ring configuration consisting entirely of NT nodes is a refinement of the configuration consisting in just one NT node; second, an open-ring configuration with any number of NT nodes and one T node is a refinement of a configuration consisting in a single T node. Our implementation is able to perform the above compatibility and refinement checks in a fraction of a second. ∎

The notion of refinement, in addition to implementation, captures also substitutivity: if N refines M, and M is compatible with the remainder P of the design, then P is also compatible with N.

Theorem 2.2 (Substitutivity of refinement) *Consider three bidirectional Moore or bidirectional interfaces M, N, P, such that $M \between P$, and $N \preceq M$. If $(\mathcal{V}_N^o \cap \mathcal{V}_P^i) \subseteq (\mathcal{V}_M^o \cap \mathcal{V}_P^i)$, then $N \between P$ and $(N \| P) \preceq (M \| P)$.*

The result has a proviso: all the variables that are output by N and input by P should also be output by M. If this were not the case, it would be possible for the additional outputs of N to violate the input assumptions of P.

2.4 Compositional Verification

The goal of compositional methods is to prove that an implementation satisfies a specification by reasoning separately on the various components of the implementation. When the components M and N of a design are implemented as M' and N', to prove the correctness of the implementation it is necessary to prove the refinement $(M' \| N) \preceq (M \| N)$. The simplest approach to proving this refinement consists in proving separately that $M' \preceq M$ and $N' \preceq N$. Unfortunately, this approach seldom works: usually, the implementation M' refines the specification M only when it receives suitable inputs from an appropriate environment such as, hopefully, N'; similarly for N' and N. Various improvements to this basic rule have been proposed, based on the idea of using an environment $E_{M'}$ to restrict the inputs to M', and proving $M' \| E_{M'} \preceq M$ (and symmetrically for N'). For instance, the following two

rules have been proposed:

$$\frac{M' \| N \preceq M; \quad N' \| M \preceq N}{M' \| N' \preceq M \| N} \quad \text{(AG-SPEC)}$$

$$\frac{M' \| E_{M'} \preceq M; \quad N' \| E_{N'} \preceq N}{M' \| N' \preceq M \| N} \quad \text{(AG-IMPL)}$$

In Rule (AG-SPEC), the environment for M' is the specification N [17, 90]; in Rule (AG-IMPL), the environment $E_{M'}$ for M' describes N' and is either provided by the user, or is obtained with approximate automatic methods [16]. Since interfaces can represent input restrictions directly, instead of providing an environment $E_{M'}$ for M', we can restrict the input assumptions of M' to reflect the inputs that can occur in its actual environment N'. Given two compatible interfaces M' and N', we call the *adaptation of M' to the environment N'* a strengthening of the input assumptions of M' that reflects the inputs that M' can receive from N' in $M' \| N'$.

Definition 2.7 (adaptation to environment) Given two compatible interfaces M and N, an *adaptation of M to N* is any interface $\widehat{M}$ that differs from M only for the input assumptions, and such that (i) $M \preceq \widehat{M}$, (ii) $\widehat{M} \wr\wr N$ iff $M \wr\wr M$, and (iii) if $M \wr\wr N$, then for all interfaces P, we have that $(\widehat{M} \| N) \wr\wr P$ iff $(M \| N) \wr\wr P$.

Condition (i) ensures that the input invariants of $\widehat{M}$ do not weaken those of M. Conditions (ii) and (iii) together require that $M \| N \| P$ is defined iff $\widehat{M} \| N \| P$ is defined: hence, the input strengthening cannot rule out any input that M will receive when in the environment of N. There are several ways of adapting an interface M to its environment N. The following proposition describes a method for Moore interfaces that is informed by the technique of [16] for the automated construction of environments. Given a set $\mathcal{V}$ of variables, a predicate θ on $\mathcal{V}$, and a predicate τ on $\mathcal{V} \cup \mathcal{V}'$, define $\mathit{Reach}(\mathcal{V}, \theta, \tau)$ to be the fixpoint of the reachability computation

$R_0 = \theta$, and $R'_{k+1} = R'_k \vee \exists \mathcal{V}.(R_k \wedge \tau)$, for $k \geq 0$. The proposition strengthens the input assumptions of M on the basis of both the output *and the input* behavior of N.

Proposition 2.1 *Given two Moore interfaces M and N, let $\mathcal{U} = \mathcal{V}_N \setminus \mathcal{V}_M$, and let $\widehat{M}$ be obtained from M by replacing θ^i_M with $\theta^i_M \wedge \exists \mathcal{U}.\theta^o_N$, and by replacing τ^i_M with $\tau^i_M \wedge \exists(\mathcal{U} \cup \mathcal{U}').(Reach(\mathcal{V}_N, \theta^o_N \wedge \theta^i_N, \tau^o_N \wedge \tau^i_N) \wedge \tau^o_N)$. Then, $\widehat{M}$ is an adaptation of M to N.*

The subtlety of the proposition lies in the fact that, to adapt M, we use a reachability predicate for N that is computed under the assumption that the input assumptions of N are respected, even though M itself may violate them. The correctness of the proposition depends on the fact that, if M violates the input assumptions of N, then so does the adaptation $\widehat{M}$, so that in Definition 2.7 it will be neither $M \wr\wr N$ nor $\widehat{M} \wr\wr N$. The following theorem states that, in proving a compositional refinement, we can adapt each interface to its environment. In the theorem, and in the discussion below, we use the notation $M_{[N]}$ to denote an interface that is an adaptation of M to N.

Theorem 2.3 (compositional refinement and adaptation) *For all interfaces M, N, M', N' such that (i) $M \wr\wr N$, (ii) M' and N' are composable (but not necessarily compatible), (iii) $(\mathcal{V}^o_{M'} \cap \mathcal{V}^i_N) \subseteq (\mathcal{V}^o_M \cap \mathcal{V}^i_N)$, and (iv) $(\mathcal{V}^o_{N'} \cap \mathcal{V}^i_M) \subseteq (\mathcal{V}^o_N \cap \mathcal{V}^i_M)$, the following verification rule is valid:*

$$\frac{M'_{[N']} \preceq M_{[N]}; \quad N'_{[M']} \preceq N_{[M]}}{M' \wr\wr N' \text{ and } \quad M' \| N' \preceq M \| N} \quad \text{(AG-INTF)}$$

The proviso over the variables is necessary to ensure that the compatibility of M and N implies that of M' and N', once the refinement is proved. The theorem states

a circular assume-guarantee principle: in fact, as illustrated by Proposition 2.1, the adaptation of M' on N' can be constructed using the fact that the input assumptions of N' are respected, even though the compatibility of M' and N' is a conclusion of the rule, rather than a premise. It is easy to see that rule (AG-INTF) generalizes (AG-IMPL); to see that it also generalizes (AG-SPEC), note that rule (AG-SPEC) can be restated as $(M' \preceq M_{[N]}; N' \preceq N_{[M]})/(M'\|N' \preceq M\|N)$. In fact, our alternating notion of refinement requires the implementation to match the specification only when the implementation is subject to the same inputs as the specification. Hence, if we restrict the input assumptions of the specification (as in $M_{[N]}$), we need to check that the refinement holds only when the implementation M' is subject to similarly restricted inputs: thus, checking the alternating refinement $M' \preceq M_{[N]}$ corresponds to checking the regular refinement $M'\|N \preceq M$ (where M, N, and M' are interpreted as regular modules, rather than interfaces). A symmetrical argument holds for the other premise. Thus, Theorem 2.3 highlights how the essence of compositional refinement checking lies in studying both implementation and specification components adapted to their actual environment.

Acknowledgements

The work reported in this chapter was conducted jointly with Prof Luca de Alfaro, Prof Thomas A. Henzinger, and Dr Freddy Y. C. Mang. This research was funded by Prof Henzinger supported in part by the AFOSR grant F49620-00-1-0327, the DARPA grant F33615-00-C-1693, the MARCO grant 98-DT-660, the NSF grant CCR-9988172, the SRC grant 99-TJ-683.003, and the NSF CAREER award CCR-

0132780. This chapter is based on a paper [41] presented at CAV 2002, copyright held by Springer-Verlag Berlin Heidelberg[1], 2002.

[1] http://www.springerlink.com

Chapter 3

An Interface Formalism for Web Services

Web application development using distributed components and web services presents new software integration challenges, because solutions often cross vendor, administrative, and other boundaries across which neither binary nor source code can be shared. We present a methodology that addresses this problem through a formalism for specifying and manipulating behavioral interfaces of multi-threaded open software components that communicate with each other through method calls. An interface constrains both the implementation and the user of a web service to fulfill certain assumptions that are specified by the interface. Our methodology consists of three increasingly expressive classes of interfaces. *Signature interfaces* specify the methods that can be invoked by the user, together with parameters. *Consistency interfaces* add propositional constraints, enhancing signature interfaces with the ability to specify choice and causality. *Protocol interfaces* specify, in addition, temporal ordering constraints on method invocations. We provide approaches to check if two

or more interfaces are *compatible*; if a web service can be safely *substituted* for another one; and if a web service *satisfies* a *specification* that represents a desired behavioral property.

3.1 Introduction

Component-based design for complex software systems has been an area of active interest for some time. Building web applications using distributed components or web services introduces special challenges. Conventional development of a software product is often done by a single vendor; and each developer has access to the entire source code or can use debugging tools on an executable built from all the software that his own contribution needs to interact with. In contrast, a web application often uses services offered by a number of different service providers, most of which do not disclose even their executable binaries, leave alone the source code; and the web application developer using these services has to rely solely on the disclosed documentation, which is usually informal, ambiguous, and often self-contradictory. In such a situation, *interface formalisms* provide a means for unambiguously describing and manipulating constraints under which independently developed software components can work properly together.

Static type systems used in programming languages constitute a simple interface formalism to avoid composition errors: a function's signature ensures, for example, that the function is only called with the correct number of parameters and that the parameters are of the correct type. Richer interface formalisms for software components have been proposed for communication protocols [54], timing requirements [57], and resource usage [42]. In the spirit of these interface theories, we present a formalism for

web service interfaces which supports a two-player game view of multi-threaded open software components. This view makes the formalism applicable to scenarios where the details of the concrete implementation of a service, as well as the details of the environment of the service, are not known at design and analysis time. However, an interface constrains both the implementation and the environment of the service with assumptions that are made by the designer of the service. This enables our approach to be used early in the web software design cycle, and by service developers who do not have access to the source or binary code of partner services that form part of their environment. A preliminary version of our interfaces [25] did not support the two-player view, permitting analysis only for closed systems, where the environment is known.

In contrast to the formal verification of web service *implementations* [74, 75, 72, 120], we explicitly propose to specify and verify *interfaces* of web services, which has a much better chance of succeeding in practice, because interfaces are usually less complex than the corresponding implementations. Indeed, good interface design requires that an interface exposes all information needed to use the service properly, but no more. In this spirit, we present three interface description languages of successively increasing expressiveness. Using these languages, we can automatically check if two or more interfaces are *compatible* (i.e., if they satisfy each other's assumptions), and if one interface can be safely *substituted* for another one in every design. In addition to this, we introduce *specifications* to describe desired propositional and temporal properties of web services, and we provide means for checking whether an interface satisfies them.

The first formalism, called *signature interfaces*, exposes only the names and types of web methods provided by the service, and the names and types of web methods that

the interface expects to be provided by the environment. For example, a signature interface may offer the web method `ProcPay`, with the two possible return values `OK` and `FAIL`, and it may itself rely on certain other methods and return values. The second formalism, called *consistency interfaces*, adds propositional constraints representing choice and causality to signature interfaces; for example, it may prohibit having both an invocation of `ProcPay` with return value `FAIL` and invocation of `ShipItem` with return value `OK` in the same conversation. The third and richest formalism, called *protocol interfaces*, adds temporal ordering constraints to consistency interfaces; for example, it may disallow conversations where `ShipItem` is invoked with return value `OK` before `ProcPay` is invoked with return value `OK`.

Example 3.1 (Supply chain management application) In the following sections, we use a simple example to illustrate the introduced interfaces. The supply chain management application consists of five web services: `Shop`, `Store`, `Bank`, `Transport`, and `Supplier`. Figure 3.1 shows an architectural overview of the application. Labeled arrows from one service to another indicate web method calls from caller to callee. `Shop` supports the web method `SellItem` called by `Client` to start the selling process, and `ChkAvail` which checks availability of items to be sold and is called by `Shop` itself. `Shop` requires the web method `ChkStore` implemented by `Store` to check whether desired items are in stock. It also requires `ShipItem` implemented by `Transport` to ship items to the customer, and `ProcPay` implemented by `Bank` to process credit card payments. `Store` requires `GetOffer` and `Order` implemented by `Supplier` to get an offer for and order new items respectively. ∎

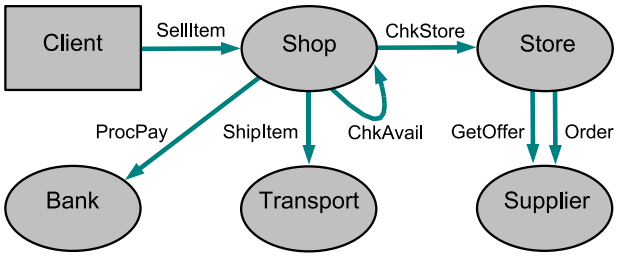


Figure 3.1. The supply chain management application

3.2 Signature Interfaces

Let $\mathcal{M}$ and $\mathcal{I}$ be finite sets of *web methods* and *instances*, respectively. Instances are associated with calls to web methods, and encode parameters passed to the web method, return values from the web method, and other behavioral differences between various calls to the web method; for instance, if the invocation was synchronous or asynchronous, or in the latter case, if it will lead to a callback. A *namespace* is a set $\mathcal{N} \subseteq \mathcal{M}$. Let $\mathcal{A} \subseteq \mathcal{M} \times \mathcal{I}$ be the set of *actions*. The web method associated with an action a is denoted as $[a]$. Given $A \subseteq \mathcal{A}$, $[A]$ denotes $\{[a] \mid a \in A\}$.

A *signature* $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ consists of a namespace $\mathcal{N}$, a set $\mathcal{J} \subseteq \mathcal{A}$ of actions that are *supported by* $\mathcal{S}$ such that $[\mathcal{J}] \subseteq \mathcal{N}$, a set $\mathcal{K} \subseteq \mathcal{A}$ of *external* actions that are *required by* $\mathcal{S}$ such that $[\mathcal{K}] \cap \mathcal{N} = \emptyset$, and a partial function $\mathcal{D} : \mathcal{J} \to 2^{\mathcal{A}}$ which assigns to a supported action a a set of actions that can be (directly) invoked by a. A signature $\mathcal{S}$ *supports* a web method $m \in \mathcal{M}$ if $\mathcal{S}$ supports an action a such that $[a] = m$. An action a *requires* an action a' in $\mathcal{S}$ if $a' \in \mathcal{D}(a)$. A signature $\mathcal{S}$ *requires* an action a' if some action a requires a' in $\mathcal{S}$. A signature $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ is a *signature interface* if $\mathcal{D}(a) \subseteq (\mathcal{N} \times \mathcal{I}) \cup \mathcal{K}$ for all $a \in \mathcal{J}$.

Intuitively, an element $(\langle m, i \rangle, D)$ of $\mathcal{D}$ says that when the web method m is called and the caller assumes the instance i, the signature $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ pledges

to support this action, and itself relies on that the assumptions carried by the actions $a' \in D$ are fulfilled (by either this signature, or by the environment, or by a refinement of this signature). Thus, a signature interface relates the *"guarantees"* made (actions supported) by the interface to the *"assumptions"* (actions assumed to be supported, either by the interface itself or by the environment) under which they are made.

Example 3.2 (Signature interface) The signature interface for `Shop` uses the following sets of web methods, instances, and actions:

$$\begin{aligned}
\mathcal{M} &= \{ \ \texttt{SellItem}, \texttt{ChkAvail}, \texttt{ChkStore}, \texttt{ProcPay}, \texttt{ShipItem}, \texttt{GetOffer}, \texttt{Order} \ \} \\
\mathcal{I} &= \{ \ \texttt{SOLD}, \texttt{NOTFOUND}, \texttt{OK}, \texttt{FAIL}, \texttt{REC} \ \} \\
\mathcal{A} &= \{ \ \langle \texttt{SellItem}, \texttt{SOLD} \rangle, \langle \texttt{SellItem}, \texttt{FAIL} \rangle, \langle \texttt{SellItem}, \texttt{NOTFOUND} \rangle, \\
&\quad \langle \texttt{ChkAvail}, \texttt{OK} \rangle, \langle \texttt{ChkAvail}, \texttt{FAIL} \rangle, \langle \texttt{ChkStore}, \texttt{OK} \rangle, \langle \texttt{ChkStore}, \texttt{FAIL} \rangle, \\
&\quad \langle \texttt{ProcPay}, \texttt{OK} \rangle, \langle \texttt{ProcPay}, \texttt{FAIL} \rangle, \langle \texttt{ShipItem}, \texttt{OK} \rangle, \langle \texttt{ShipItem}, \texttt{FAIL} \rangle, \\
&\quad \langle \texttt{GetOffer}, \texttt{REC} \rangle, \langle \texttt{Order}, \texttt{OK} \rangle \ \}
\end{aligned}$$

Now we can define a signature $\mathcal{S}_{\texttt{Shop}}$ consisting of the following components:

$$\begin{aligned}
\mathcal{N}_{\texttt{Shop}} &= \{ \ \texttt{SellItem}, \texttt{CheckAvail} \ \} \\
\mathcal{J}_{\texttt{Shop}} &= \{ \ \langle \texttt{SellItem}, \texttt{SOLD} \rangle, \langle \texttt{SellItem}, \texttt{FAIL} \rangle, \langle \texttt{ChkAvail}, \texttt{OK} \rangle, \langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \} \\
\mathcal{K}_{\texttt{Shop}} &= \{ \ \langle \texttt{ChkStore}, \texttt{OK} \rangle, \langle \texttt{ChkStore}, \texttt{FAIL} \rangle, \langle \texttt{ProcPay}, \texttt{OK} \rangle, \langle \texttt{ProcPay}, \texttt{FAIL} \rangle, \\
&\quad \langle \texttt{ShipItem}, \texttt{OK} \rangle, \langle \texttt{ShipItem}, \texttt{FAIL} \rangle \ \}
\end{aligned}$$

$\mathcal{D}_{\texttt{Shop}} = \{$

$$
\begin{aligned}
&\langle \texttt{SellItem}, \texttt{SOLD} \rangle \mapsto \{\langle \texttt{ChkAvail}, \texttt{OK} \rangle, \langle \texttt{ProcPay}, \texttt{OK} \rangle, \langle \texttt{ShipItem}, \texttt{OK} \rangle\} \\
&\langle \texttt{SellItem}, \texttt{FAIL} \rangle \mapsto \{\langle \texttt{ChkAvail}, \texttt{OK} \rangle, \langle \texttt{ChkAvail}, \texttt{FAIL} \rangle, \\
&\qquad\qquad \langle \texttt{ProcPay}, \texttt{OK} \rangle, \langle \texttt{ProcPay}, \texttt{FAIL} \rangle, \langle \texttt{ShipItem}, \texttt{FAIL} \rangle\} \\
&\langle \texttt{ChkAvail}, \texttt{OK} \rangle \mapsto \{\langle \texttt{ChkStore}, \texttt{OK} \rangle\} \\
&\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \mapsto \{\langle \texttt{ChkStore}, \texttt{FAIL} \rangle\} \\
\}
\end{aligned}
$$

For instance, action $\langle \texttt{SellItem}, \texttt{SOLD} \rangle$ is supported by $\mathcal{S}_{\texttt{Shop}}$, and actions $\langle \texttt{ChkAvail}, \texttt{OK} \rangle, \langle \texttt{ProcPay}, \texttt{OK} \rangle$ and $\langle \texttt{ShipItem}, \texttt{OK} \rangle$ are assumed to be supported by the environment. The action $\langle \texttt{SellItem}, \texttt{NOTFOUND} \rangle$ is not supported, but belongs to the namespace of the signature — it could be supported in a refinement of this signature.

Signature $\mathcal{S}_{\texttt{Shop}}$ is a signature interface, because all actions it uses are from the local namespace $\mathcal{N}$ or from the set of environment actions $\mathcal{K}$. ∎

3.2.1 Compatibility and Composition

Given two signature interfaces $\mathcal{S}_1 = (\mathcal{N}_1, \mathcal{J}_1, \mathcal{K}_1, \mathcal{D}_1)$ and $\mathcal{S}_2 = (\mathcal{N}_2, \mathcal{J}_2, \mathcal{K}_2, \mathcal{D}_2)$, if $\mathcal{N}_1 \cap \mathcal{N}_2 = \emptyset$, then $\mathcal{S}_1$ and $\mathcal{S}_2$ are *compatible* (denoted by $comp(\mathcal{S}_1, \mathcal{S}_2)$), and their *composition* (denoted by $(\mathcal{S}_1 \parallel \mathcal{S}_2)$) is $\mathcal{S}_c = (\mathcal{N}_c, \mathcal{J}_c, \mathcal{K}_c, \mathcal{D}_c)$, where $\mathcal{N}_c = \mathcal{N}_1 \cup \mathcal{N}_2$, and $\mathcal{J}_c = \mathcal{J}_1 \cup \mathcal{J}_2$, and $\mathcal{K}_c = (\mathcal{K}_1 \cup \mathcal{K}_2) \setminus (\mathcal{N}_c \times \mathcal{I})$, and $\mathcal{D}_c = \mathcal{D}_1 \cup \mathcal{D}_2$. The composition operation is commutative and associative. Compatibility and composition of signature interfaces can be computed in $O(n \cdot \log n)$ time, where $n = \max(|\mathcal{N}_1|, |\mathcal{N}_2|)$.

A signature interface $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ is *closed* if it requires only actions a with $[a] \in \mathcal{N}$. A signature interface is *open* if it is not closed. A signature interface $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ is *concrete* if it supports all actions a with $[a] \in \mathcal{N}$. A signature

is *abstract* if it is not concrete. Given a signature interface $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$, an *environment* for $\mathcal{S}$ is a concrete signature interface $\mathcal{E}$ that is compatible with $\mathcal{S}$ such that the composition $\mathcal{S} \parallel \mathcal{E}$ is closed. Note that $\mathcal{S} \parallel \mathcal{E}$ is concrete if and only if $\mathcal{S}$ is concrete, and $\mathcal{E}$ is not unique. Intuitively, each $\mathcal{E}$ represents a design context in which $\mathcal{S}$ can be used.

3.2.2 Refinement

Given two signature interfaces $\mathcal{S} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$ and $\mathcal{S}' = (\mathcal{N}', \mathcal{J}', \mathcal{K}', \mathcal{D}')$, we say $\mathcal{S}'$ *refines* $\mathcal{S}$ (written $\mathcal{S}' \preccurlyeq \mathcal{S}$) if (i) $\mathcal{N}' \subseteq \mathcal{N}$, (ii) $\mathcal{J}' \supseteq \mathcal{J}$, (iii) $\mathcal{K}' \subseteq \mathcal{K}$, and (iv) for every $a \in \mathcal{J}$, if a requires a' in $\mathcal{S}'$, then a requires a' in $\mathcal{S}$.

The first condition ensures that the refined signature interface does not try to reserve additional names for itself. The second condition ensures that the refined signature interface *guarantees* to support every action that is supported by the abstract one. The other two conditions ensure that the refined signature interface does not *assume* additional actions to be supported by the environment. Given two signature interfaces $\mathcal{S}$ and $\mathcal{S}'$, the question if $\mathcal{S}' \preccurlyeq \mathcal{S}$ can be decided in $O(n \cdot \log n)$ time, where $n = \max(|\mathcal{N}|, |\mathcal{N}'|)$.

Note that the above definition leads to *substitutivity of refinements:* for signature interfaces $\mathcal{S}_1$, $\mathcal{S}_1'$ and $\mathcal{S}_2$, if $comp(\mathcal{S}_1, \mathcal{S}_2)$ and $\mathcal{S}_1' \preccurlyeq \mathcal{S}_1$, then $comp(\mathcal{S}_1', \mathcal{S}_2)$ and $\mathcal{S}_1' \parallel \mathcal{S}_2 \preccurlyeq \mathcal{S}_1 \parallel \mathcal{S}_2$. Intuitively, this means that in an overall design, an abstract placeholder component can be safely replaced with a refined version of it, and the overall design would not exhibit any incorrect behavior if it did not do so before.

3.3 Consistency Interfaces

A *consistency interface* $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$ consists of a namespace $\mathcal{N}$, a set $\mathcal{J} \subseteq \mathcal{A}$ of actions that are *supported by* $\mathcal{C}$ such that $[\mathcal{J}] \subseteq \mathcal{N}$, a set $\mathcal{K} \subseteq \mathcal{A}$ of *external* actions that are *required by* $\mathcal{C}$ such that $[\mathcal{K}] \cap \mathcal{N} = \emptyset$, and a partial function $\mathcal{F} : \mathcal{A} \to \mathcal{B}(\mathcal{A})$, which assigns to a supported action a an expression from $\mathcal{B}(\mathcal{A})$, the set of expressions over the set of actions $\mathcal{A}$ using the binary operators $\sqcap$ and $\sqcup$, and the constant ϵ.

Given a consistency interface $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$, the *underlying signature* of $\mathcal{C}$ (denoted by $sig(\mathcal{C})$) is $(\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$, where $\mathcal{D} : \mathcal{A} \to 2^{\mathcal{A}}$ is defined as follows: for all $a \in \mathcal{J}$, $\mathcal{D}(a) = \{a' \mid a' \text{ occurs in expression } \mathcal{F}(a)\}$. We require that the underlying signature of a consistency interface is a signature interface.

Example 3.3 (Consistency interface) Now we model `Shop` as a consistency interface $\mathcal{C}_{\texttt{Shop}} = (\mathcal{N}_{\texttt{Shop}}, \mathcal{J}_{\texttt{Shop}}, \mathcal{K}_{\texttt{Shop}}, \mathcal{F}_{\texttt{Shop}})$ where $\mathcal{N}_{\texttt{Shop}}$, and $\mathcal{J}_{\texttt{Shop}}$, and $\mathcal{K}_{\texttt{Shop}}$ are as in Example 3.2, and $\mathcal{F}_{\texttt{Shop}}$ is as follows:

$$
\begin{aligned}
&\mathcal{F}_{\texttt{Shop}} = \{ \\
&\langle \texttt{SellItem}, \texttt{SOLD} \rangle \mapsto \quad \langle \texttt{ChkAvail}, \texttt{OK} \rangle \sqcap \langle \texttt{ProcPay}, \texttt{OK} \rangle \sqcap \langle \texttt{ShipItem}, \texttt{OK} \rangle \\
&\langle \texttt{SellItem}, \texttt{FAIL} \rangle \mapsto \quad \langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \sqcup \\
&\qquad\qquad\qquad\qquad (\langle \texttt{ChkAvail}, \texttt{OK} \rangle \ \sqcap \ (\langle \texttt{ProcPay}, \texttt{FAIL} \rangle \sqcup \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\langle \texttt{ProcPay}, \texttt{OK} \rangle \sqcap \langle \texttt{ShipItem}, \texttt{FAIL} \rangle) \ ) \ ) \\
&\langle \texttt{ChkAvail}, \texttt{OK} \rangle \mapsto \quad \langle \texttt{ChkStore}, \texttt{OK} \rangle \\
&\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \mapsto \quad \langle \texttt{ChkStore}, \texttt{FAIL} \rangle \\
&\}
\end{aligned}
$$

This consistency interface is a natural extension of the signature interface $\mathcal{S}_{\texttt{Shop}}$; it keeps different choices in the conversations separate.

For action $\langle \texttt{SellItem}, \texttt{SOLD} \rangle$, all three actions in the expression on the right hand

side occur together. For action $\langle \texttt{SellItem}, \texttt{FAIL} \rangle$, action $\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle$ occurs alone, or action $\langle \texttt{ChkAvail}, \texttt{OK} \rangle$ occurs together with either action $\langle \texttt{ProcPay}, \texttt{FAIL} \rangle$ or both, actions $\langle \texttt{ProcPay}, \texttt{OK} \rangle$ and $\langle \texttt{ShipItem}, \texttt{FAIL} \rangle$. Notice that nothing is said about the order of their occurrence. The actions for method `ChkAvail` result in calls to the method `ChkStore` in `Store`.

The underlying signature of $\mathcal{C}_{\texttt{Shop}}$ is $\mathcal{S}_{\texttt{Shop}}$ from the example in the last section. Our $\mathcal{C}_{\texttt{Shop}}$ is a consistency interface because its underlying signature is a signature interface. ∎

3.3.1 Compatibility and Composition

Given two consistency interfaces $\mathcal{C}_1 = (\mathcal{N}_1, \mathcal{J}_1, \mathcal{K}_1, \mathcal{F}_1)$ and $\mathcal{C}_2 = (\mathcal{N}_2, \mathcal{J}_2, \mathcal{K}_2, \mathcal{F}_2)$, if the underlying signatures $sig(\mathcal{C}_1)$ and $sig(\mathcal{C}_2)$ are compatible, then $\mathcal{C}_1$ and $\mathcal{C}_2$ are *compatible* (denoted by $comp(\mathcal{C}_1, \mathcal{C}_2)$), and their *composition* (denoted $\mathcal{C}_1 \parallel \mathcal{C}_2$) is $\mathcal{C}_c = (\mathcal{N}_c, \mathcal{J}_c, \mathcal{K}_c, \mathcal{F}_c)$ where $\mathcal{N}_c = \mathcal{N}_1 \cup \mathcal{N}_2$, and $\mathcal{J}_c = \mathcal{J}_1 \cup \mathcal{J}_2$, and $\mathcal{K}_c = (\mathcal{K}_1 \cup \mathcal{K}_2) \setminus (\mathcal{N}_c \times \mathcal{I})$, and $\mathcal{F}_c = \mathcal{F}_1 \cup \mathcal{F}_2$. The composition operation is commutative and associative. Compatibility and composition of consistency interfaces can be computed in $O(n \cdot \log n)$ time, where $n = \max(|\mathcal{N}_1|, |\mathcal{N}_2|)$. Note that the operators *sig* and $\parallel$ commute: for all consistency interfaces $\mathcal{C}_1$ and $\mathcal{C}_2$, we have $sig(\mathcal{C}_1 \parallel \mathcal{C}_2) = sig(\mathcal{C}_1) \parallel sig(\mathcal{C}_2)$.

A consistency interface $\mathcal{C}$ is *closed* (*concrete*) if $sig(\mathcal{C})$ is closed (concrete). Given a consistency interface $\mathcal{C}$, an *environment* for $\mathcal{C}$ is a concrete consistency interface $\mathcal{E}$ that is compatible with $\mathcal{C}$, such that the composition $\mathcal{C} \parallel \mathcal{E}$ is closed.

3.3.2 Refinement

A *conversation* of a consistency interface is a set of actions that are exhibited together in one execution of the system. Given a consistency interface $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$, the set of conversations represented by an expression from $\mathcal{B}(\mathcal{A})$ is defined by the function $[\![.]\!] : \mathcal{B}(\mathcal{A}) \to 2^{2^{\mathcal{A}}}$, which is defined as the least solution of the following system of equations, where $a \in \mathcal{A}$ and $\varphi_1, \varphi_2 \in \mathcal{B}(\mathcal{A})$:

$$
\begin{array}{lcll}
[\![\mathrm{T}]\!] & = & \{\{\}\} & \\
[\![a]\!] & = & \{\{a\} \cup y \mid y \in [\![\mathcal{F}(a)]\!]\} & \text{if } a \in \mathcal{J} \\
[\![a]\!] & = & \{\{a\} \cup y \mid y \subseteq (\mathcal{N} \times \mathcal{I}) \cup \mathcal{K}\} & \text{if } a \notin \mathcal{J} \text{ but } [a] \in \mathcal{N} \\
[\![a]\!] & = & \{\{a\}\} & \text{if } [a] \notin \mathcal{N} \\
[\![\varphi_1 \sqcup \varphi_2]\!] & = & [\![\varphi_1]\!] \cup [\![\varphi_2]\!] & \\
[\![\varphi_1 \sqcap \varphi_2]\!] & = & \{x \cup y \mid x \in [\![\varphi_1]\!], y \in [\![\varphi_2]\!]\} &
\end{array}
$$

Given two consistency interfaces $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$ and $\mathcal{C}' = (\mathcal{N}', \mathcal{J}', \mathcal{K}', \mathcal{F}')$, we say $\mathcal{C}'$ *refines* $\mathcal{C}$ (written $\mathcal{C}' \preccurlyeq \mathcal{C}$) if

i) $sig(\mathcal{C}') \preccurlyeq sig(\mathcal{C})$, and

ii) for every $a \in \mathcal{J}$, for all conversations $x' \in [\![\mathcal{F}'(a)]\!]$, there exists a conversation $x \in [\![\mathcal{F}(a)]\!]$ such that $x' \subseteq x$.

The definition above allows the refinement $\mathcal{C}'$ to drop conversations, or actions from a conversation, for actions supported by $\mathcal{C}$. The refinement $\mathcal{C}'$ is allowed to support additional actions that $\mathcal{C}$ does not support, but it is not allowed to require additional actions, and it is not allowed to introduce a new conversation x' in $\mathcal{C}'$ for an action supported by $\mathcal{C}$, such that x' is not a fragment (subset) of a conversation x of the same action in $\mathcal{C}$. The refinement-checking problem for consistency interfaces is in NP.

Theorem 3.1 *Let $\mathcal{C}_1$, $\mathcal{C}'_1$ and $\mathcal{C}_2$ be three consistency interfaces such that $\mathcal{C}'_1 \preccurlyeq \mathcal{C}_1$ and $comp(\mathcal{C}_1, \mathcal{C}_2)$. Then $comp(\mathcal{C}'_1, \mathcal{C}_2)$ and $\mathcal{C}'_1 \parallel \mathcal{C}_2 \preccurlyeq \mathcal{C}_1 \parallel \mathcal{C}_2$.*

Proof.

i) Given $\mathcal{C}'_1 \preccurlyeq \mathcal{C}_1$, we get, by definition of refinement for consistency interfaces, $sig(\mathcal{C}'_1) \preccurlyeq sig(\mathcal{C}_1)$. Then, by definition of refinement for signature interfaces, $\mathcal{N}'_1 \subseteq \mathcal{N}_1$, where $\mathcal{N}'_1$ and $\mathcal{N}_1$ are the namespaces of signature interfaces $sig(\mathcal{C}'_1)$ and $sig(\mathcal{C}_1)$ respectively.

Now, given $comp(\mathcal{C}_1, \mathcal{C}_2)$, we have, by definition of compatibility of consistency interfaces, $comp(sig(\mathcal{C}_1), sig(\mathcal{C}_2))$. Then, by definition of compatibility for signature interfaces, we have $\mathcal{N}_1 \cap \mathcal{N}_2 = \emptyset$, where $\mathcal{N}_1$ and $\mathcal{N}_2$ are the namespaces of the signature interfaces $sig(\mathcal{C}_1)$ and $sig(\mathcal{C}_2)$ respectively.

From the above, we have $\mathcal{N}'_1 \cap \mathcal{N}_2 = \emptyset$, and thence, we get $comp(sig(\mathcal{C}'_1), sig(\mathcal{C}_2))$, and thus, we conclude $comp(\mathcal{C}'_1, \mathcal{C}_2)$.

ii) Let us assume that the required result does not hold. In other words, let $\mathcal{C}'_1 \preccurlyeq \mathcal{C}_1$, but $(\mathcal{C}'_1 \parallel \mathcal{C}_2) \not\preccurlyeq (\mathcal{C}_1 \parallel \mathcal{C}_2)$. Note that by definitions of composition for consistency interfaces, underlying signatures for consistency interfaces, and refinement for signature interfaces, we have $sig(\mathcal{C}'_1 \parallel \mathcal{C}_2) \preccurlyeq sig(\mathcal{C}_1 \parallel \mathcal{C}_2)$. Thus, by definition of refinement for consistency interfaces, and by our assumption $(\mathcal{C}'_1 \parallel \mathcal{C}_2) \not\preccurlyeq (\mathcal{C}_1 \parallel \mathcal{C}_2)$, we conclude that there must be at least one action $a \in \mathcal{J}_1 \cup \mathcal{J}_2$, where $\mathcal{C}_i = (\mathcal{N}_i, \mathcal{J}_i, \mathcal{K}_i, \mathcal{F}_i)$ for $i \in \{1, 2\}$ and $\mathcal{C}'_1 = (\mathcal{N}'_1, \mathcal{J}'_1, \mathcal{K}'_1, \mathcal{F}'_1)$, such that there exists a conversation $y \in [\![\mathcal{F}'(a)]\!]$, such that for all conversations $x \in [\![\mathcal{F}(a)]\!]$ we have $y \not\subseteq x$, where $\mathcal{F}' = \mathcal{F}'_1 \cup \mathcal{F}_2$ and $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$. To make this happen, the only way is for an action supported by $\mathcal{C}'_1$ but not by $\mathcal{C}_1$ to invoke additional actions. However, since $\mathcal{K}'_1 \subseteq \mathcal{K}_1$ by definition of refinement

for consistency interfaces (and for signature interfaces), and by the definition of conversations represented by formulas from $\mathcal{B}(\mathcal{A})$ on consistency interfaces as defined above, we see that this is not possible. Thus, a contradiction is reached, and the required result follows. ∎

3.3.3 Specifications

A *specification* ψ for a consistency interface is a formula $a \not\rightsquigarrow S$ where $a \in \mathcal{A}$ and $S \subseteq \mathcal{A}$. Intuitively, a specification $a \not\rightsquigarrow S$ states that the invocation of action a must not lead to a conversation in which the actions in set S are all exhibited together. Formally, a specification $\psi = a \not\rightsquigarrow S$ *is satisfied by* a consistency interface $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$ (denoted $\mathcal{C} \models \psi$) if $S \not\subseteq x$ for all $x \in [\![\mathcal{F}(a)]\!]$. The specification satisfaction problem for consistency interfaces is in co-NP. Given two compatible consistency interfaces $\mathcal{C}_1$ and $\mathcal{C}_2$, and a specification ψ, the following holds: $(\mathcal{C}_1 \parallel \mathcal{C}_2) \models \psi$ implies $\mathcal{C}_1 \models \psi$ and $\mathcal{C}_2 \models \psi$. The converse is not true.

Theorem 3.2 *Given a consistency interface $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$ and a specification $\psi = a \not\rightsquigarrow S$, $\mathcal{C} \models \psi$ if and only if for all consistency interfaces $\mathcal{C}' = (\mathcal{N}', \mathcal{J}', \mathcal{K}', \mathcal{F}')$ such that $\mathcal{C}' \preccurlyeq \mathcal{C}$, there exists an environment $\mathcal{E}_{\mathcal{C}'}$ of $\mathcal{C}'$, such that $(\mathcal{C}' \parallel \mathcal{E}_{\mathcal{C}'}) \models \psi$.*

Proof.

($\Rightarrow$) Let us assume that the required result does not hold. Then, we have, given $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$ and a specification $\psi = a \not\rightsquigarrow S$, and that $\mathcal{C} \models \psi$, and by assumption, that there exists a consistency interface $\mathcal{C}' = (\mathcal{N}', \mathcal{J}', \mathcal{K}', \mathcal{F}')$ such that $\mathcal{C}' \preccurlyeq \mathcal{C}$, but for all environments $\mathcal{E}_{\mathcal{C}'}$ of $\mathcal{C}'$, $(\mathcal{C}' \parallel \mathcal{E}_{\mathcal{C}'}) \not\models \psi$. Now, we have three cases:

a) a is outside namespace $\mathcal{N}$: Then a belongs to the namespace of $\mathcal{E}_{\mathcal{C}'}$, and hence the assumption cannot occur. A contradiction is reached, and hence this subcase has to be rejected if our assumption has to be true.

b) a is in namespace $\mathcal{N}$, but $a \notin \mathcal{J}$: Then $[\![\mathcal{F}(a)]\!]$ in $\mathcal{C}$ is $\{\{a\} \cup y \mid y \subseteq (\mathcal{N} \times \mathcal{I}) \cup \mathcal{K}\}$ by definition. Then, by definition of refinement of consistency interfaces, it is not possible that there should exist a $y \in [\![\mathcal{F}'(a)]\!]$ in $\mathcal{C}'$ such that $S \subseteq y$ but $S \nsubseteq x$ for all $x \in [\![\mathcal{F}(a)]\!]$ in $\mathcal{C}$, which must hold, because we are given $\mathcal{C} \models \psi$. Thus, this subcase leads to a contradiction as well.

c) $a \in \mathcal{J}$: In this subcase, by definition of refinement for consistency interfaces, we directly have that for all conversations $y \in [\![\mathcal{F}'(a)]\!]$, there exists a conversation $x \in [\![\mathcal{F}(a)]\!]$ such that $y \subseteq x$, which rules out the possibility of existence of y such that $S \subseteq y$ and for all x, $S \nsubseteq x$. Thus, the assumption is not possible in this subcase either.

Since all possible subcases were considered, our assumption must be false, and hence the required result follows.

($\Leftarrow$) Given that for all consistency interfaces $\mathcal{C}'$ such that $\mathcal{C}' \preccurlyeq \mathcal{C}$, there exist environments $\mathcal{E}_{\mathcal{C}'}$ of $\mathcal{C}'$ such that $(\mathcal{C}' \parallel \mathcal{E}_{\mathcal{C}'}) \models \psi$, we reach, by reflexivity of the refinement relation for consistency interfaces, that $\mathcal{C}$ has an environment $\mathcal{E}_{\mathcal{C}}$ such that $(\mathcal{C} \parallel \mathcal{E}_{\mathcal{C}}) \models \psi$. Now, we observe that by definition of the semantics $[\![.]\!]$ of formulas of $\mathcal{B}(\mathcal{A})$ over consistency interfaces, for all $x \in [\![\mathcal{F}(a)]\!]$ there exists $y \in [\![(\mathcal{F} \cup \mathcal{F}^e)(a)]\!]$ such that $x \subseteq y$, where $\mathcal{E}_{\mathcal{C}} = (\mathcal{N}^e, \mathcal{J}^e, \mathcal{K}^e, \mathcal{F}^e)$. Further, we observe from $(\mathcal{C} \cup \mathcal{E}_{\mathcal{C}}) \models \psi$ and by definition of specification satisfaction for consistency interfaces, we must have $S \nsubseteq y$ for all $y \in [\![(\mathcal{F} \cup \mathcal{F}^e)(a)]\!]$, where $\psi = a \not\rightsquigarrow S$. Then, we reach $S \nsubseteq y$ for all $y \in [\![\mathcal{F}(a)]\!]$.

Now, from $(\mathcal{C} \parallel \mathcal{E}_{\mathcal{C}}) \models \psi$, we reach that either $(S \setminus ((\mathcal{N} \cup \mathcal{N}^e) \times \mathcal{I})) \not\subseteq (\mathcal{K} \cup \mathcal{K}^e)$, or, for all $b \in (\mathcal{N} \cup \mathcal{N}^e) \times \mathcal{I}$ such that $b \notin (\mathcal{J} \cup \mathcal{J}^e)$, there does not exist actions $c_0, c_1, c_2, \ldots, c_k \in (\mathcal{J} \cup \mathcal{J}^e)$ such that a requires c_0, and c_i requires c_{i+1} for $0 \leq i \leq k-1$, and c_k requires b in $sig(\mathcal{C} \parallel \mathcal{E}_{\mathcal{C}})$, where $\mathcal{C} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{F})$ and $\mathcal{E}_{\mathcal{C}} = (\mathcal{N}^e, \mathcal{J}^e, \mathcal{K}^e, \mathcal{F}^e)$. Now, $S \setminus ((\mathcal{N} \cup \mathcal{N}^e) \times \mathcal{I}) = \emptyset$, and hence the first choice is not possible. Hence, we must have: for all $b \in (\mathcal{N} \cup \mathcal{N}^e) \times \mathcal{I}$ such that $b \notin (\mathcal{J} \cup \mathcal{J}^e)$, there does not exist actions $c_0, c_1, c_2, \ldots, c_k \in \mathcal{J}$ such that a requires c_0, and c_i requires c_{i+1} for $0 \leq i \leq k-1$, and c_k requires b in $sig(\mathcal{C} \parallel \mathcal{E}_{\mathcal{C}})$. Observing that $b \in \mathcal{N} \times \mathcal{I}$ cannot satisfy $b \in \mathcal{J}^e$, we reach that for all $b \in \mathcal{N} \times \mathcal{I}$ such that $b \notin \mathcal{J}$, there does not exist actions $c_0, c_1, c_2, \ldots, c_k \in \mathcal{J}$ such that a requires c_0, and c_i requires c_{i+1} for $0 \leq i \leq k-1$, and c_k requires b in $sig(\mathcal{C})$, which is the required second condition for specification satisfaction for consistency interfaces.

From the above two paragraphs, we conclude that $\mathcal{C} \models \psi$. Therefore the required result follows. ∎

Corollary 3.1 *Let $\mathcal{C}_1$, $\mathcal{C}_1'$ and $\mathcal{C}_2$ be three consistency interfaces such that $\mathcal{C}_1' \preccurlyeq \mathcal{C}_1$, and $comp(\mathcal{C}_1, \mathcal{C}_2)$. Let ψ be a specification such that $(\mathcal{C}_1 \parallel \mathcal{C}_2) \models \psi$. Then $(\mathcal{C}_1' \parallel \mathcal{C}_2) \models \psi$.*

Proof. Let us assume that the required result does not hold. Then, a must be in the namespace of $(\mathcal{C}_1 \parallel \mathcal{C}_2)$ and a must generate a conversation x in $(\mathcal{C}_1' \parallel \mathcal{C}_2)$ such that $S \subseteq x$, where $spec = a \not\rightsquigarrow S$. Then, by definition of refinenement for consistency interfaces, since given $\mathcal{C}_1' \preccurlyeq \mathcal{C}_1$, and by Theorem 3.1, we reach that for every conversation x generated by a in $(\mathcal{C}_1' \parallel \mathcal{C}_2)$, there is a conversation y generated

by a in $\mathcal{C}_1 \parallel \mathcal{C}_2$), such that $x \subseteq y$. Then, we conclude that $(\mathcal{C}_1 \parallel \mathcal{C}_2) \not\models \psi$, which is a contradiction. Hence, our original assumption must be wrong and thus, the required result follows.

3.4 Protocol Interfaces

Let *Terms* be a set such that elements $term \in Terms$ are given by the following grammar ($a \in \mathcal{A}$ and $A \subseteq \mathcal{A}$, $|A| \geq 2$):

$$term \;::=\; \epsilon \mid a \mid \sqcap A \mid \boxplus A$$

A *protocol automaton* $\mathcal{H} = (Q, \delta)$ consists of a finite set of *locations* Q and a finite (nondeterministic) *transition relation* $\delta \subseteq Q \times Terms \times Q$, consisting of tuples $(q, term, q')$ of a *source* location q, a term *term*, and a *successor* location q'. A location that does not occur as a source location in any transition is a *return location*.

A *protocol interface* $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ consists of a namespace $\mathcal{N}$, a set $\mathcal{J} \subseteq \mathcal{A}$ of actions that are *supported by* $\mathcal{P}$ such that $[\mathcal{J}] \subseteq \mathcal{N}$, a set $\mathcal{K} \subseteq \mathcal{A}$ of *external* actions that are *required by* $\mathcal{P}$ such that $[\mathcal{K}] \cap \mathcal{N} = \emptyset$, a *protocol automaton* $\mathcal{H}$, and a partial function $\mathcal{R} : \mathcal{A} \to Q$ which assigns to a supported action a a location of $\mathcal{H}$.

The execution semantics of a protocol interface can intuitively be understood as follows. The interface maintains a set of current locations. When execution starts due to the invocation of a supported action a, this set contains exactly one location $\mathcal{R}(a)$, and execution ends when the set is empty. If a current location is q, the interface chooses a transition of $\mathcal{H}$ with q as source location and executes the term of the transition. Executing a term means: 1) for an ϵ-term, to proceed with the successor; 2) for a term of the form a, to execute the action a, and, after the execution of a is

completed, proceed with the successor; 3) for a term of the form $\sqcap A$, to execute all actions from the set A (in parallel), and, after the execution of all actions is completed, proceed with the successor; and 4) for a term of the form $\boxplus A$, to execute all actions from set A (in parallel), and, after the execution of at least one of the actions is completed, proceed with the successor. Executing an action a means adding the location $\mathcal{R}(a)$ to the set of current locations. Proceeding with the successor means the following: if the successor location of the transition is a return location, the current execution is completed; otherwise the successor location of the transition is added to the current locations.

Given a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$, the *underlying signature* of $\mathcal{P}$ (denoted $sig(\mathcal{P})$) is $(\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{D})$, where $\mathcal{D}$ is a partial function $\mathcal{D} : \mathcal{A} \rightarrow 2^{\mathcal{A}}$ such that for all $a \in \mathcal{J}$, $\mathcal{D}(a) = sigl(\mathcal{R}(a))$. The function $sigl : Q \rightarrow 2^{\mathcal{A}}$ that assigns a set of actions to every location q, is defined as follows: $sigl(q) = \bigcup_{i=0,1,\ldots,k} g(term_i) \cup \bigcup_{i=0,1,\ldots,k} sigl(q_i)$ for $(q, term_0, q_0), (q, term_1, q_1), \ldots, (q, term_k, q_k) \in \delta$. The function $g : Terms \rightarrow 2^{\mathcal{A}}$ is defined as $g(\epsilon) = \emptyset$, and $g(a) = \{a\}$, and $g(\circ A) = A$ with $\circ \in \{\sqcap, \boxplus\}$, and $a \in \mathcal{A}$, and $A \subseteq \mathcal{A}$. We require that the underlying signature of a protocol interface is a signature interface, and that starting with any location of the automaton as current location, the execution of the protocol interface can be completed.

Note that this model incorporates recursion, dynamic thread creation, and concurrency, albeit with the simplifying assumption that the concurrent threads of execution do not communicate with each other except at points of calls and returns of web method invocations. In particular, the threads of execution running in parallel are not allowed to communicate with each other through shared variables, locks, etc.

This assumption is fulfilled in the web services context: threads invoked as a result

of web method invocations through the internet usually run on physically separated remote servers (e.g. on `amazon.com`, `bankofamerica.com`, `fedex.com`, etc) and hence cannot share information through shared variables.

This is a very different (and much more simple) concurrency model compared to those typically used for verification and analysis of conventional concurrent software [60, 71, 70, 128], where patterns of communication (e.g. through shared variables) between concurrent threads (usually running on the same physical machine) is a primary issue of interest. Our model is thus not applicable in such a context. However, note that, in particular, reachability is decidable in our model, while in the conventional model for concurrent shared-memory software reachability is undecidable (since a model with at least two threads with their own separate stacks, communicating with shared variables, is equivalent to a two-counter machine). In other words, our model is constructed to take advantage of simplifying assumptions about inter-thread communication that are fulfilled in the web services context and allows us to obtain algorithmic solutions for problems that are undecidable (and hence permitting of only heuristic and semi-algorithmic solutions) in the models used in the conventional concurrent software verification and analysis context.

Example 3.4 (Protocol interface) `Shop` is represented by a protocol interface $\mathcal{P}_{\texttt{Shop}} = (\mathcal{N}_{\texttt{Shop}}, \mathcal{J}_{\texttt{Shop}}, \mathcal{K}_{\texttt{Shop}}, \mathcal{H}_{\texttt{Shop}}, \mathcal{R}_{\texttt{Shop}})$ where $\mathcal{N}_{\texttt{Shop}}$, and $\mathcal{J}_{\texttt{Shop}}$, and $\mathcal{K}_{\texttt{Shop}}$ are as in Example 3.2, and $\mathcal{H}_{\texttt{Shop}}$ and $\mathcal{R}_{\texttt{Shop}}$ are defined as follows. For readability, we define $\mathcal{H}_{\texttt{Shop}}$ and $\mathcal{R}_{\texttt{Shop}}$ by giving the transition relation δ of $\mathcal{H}_{\texttt{Shop}}$ as a sequence of tuples (q, term, q'), and indicating the partial function $\mathcal{R}$ by writing an action a in front of a transition with $\mathcal{R}(a)$ as source location (location q_0 is a return location):

$\mathcal{H}_{\texttt{Shop}}$ and $\mathcal{R}_{\texttt{Shop}}$:

$$
\begin{aligned}
\langle \texttt{SellItem}, \texttt{SOLD} \rangle \mapsto \quad & (q_1, \langle \texttt{ChkAvail}, \texttt{OK} \rangle, q_2), (q_2, \langle \texttt{ProcPay}, \texttt{OK} \rangle, q_3), \\
& (q_3, \langle \texttt{ShipItem}, \texttt{OK} \rangle, q_0), \\
\langle \texttt{SellItem}, \texttt{FAIL} \rangle \mapsto \quad & (q_4, \langle \texttt{ChkAvail}, \texttt{FAIL} \rangle, q_0), (q_4, \langle \texttt{ChkAvail}, \texttt{OK} \rangle, q_5), \\
& (q_5, \langle \texttt{ProcPay}, \texttt{FAIL} \rangle, q_0), (q_5, \langle \texttt{ProcPay}, \texttt{OK} \rangle, q_6), \\
& (q_6, \langle \texttt{ShipItem}, \texttt{FAIL} \rangle, q_0), \\
\langle \texttt{ChkAvail}, \texttt{OK} \rangle \mapsto \quad & (q_7, \langle \texttt{ChkStore}, \texttt{OK} \rangle, q_0), \\
\langle \texttt{ChkAvail}, \texttt{FAIL} \rangle \mapsto \quad & (q_8, \langle \texttt{ChkStore}, \texttt{FAIL} \rangle, q_0)
\end{aligned}
$$

This protocol interface is a natural extension of the consistency interface $\mathcal{C}_{\texttt{Shop}}$; in addition to what $\mathcal{C}_{\texttt{Shop}}$ does, $\mathcal{P}_{\texttt{Shop}}$ maintains the temporal order of actions.

It models that for action $\langle \texttt{SellItem}, \texttt{SOLD} \rangle$ the three actions occur in the given sequence in a conversation. When the action $\langle \texttt{SellItem}, \texttt{FAIL} \rangle$ is invoked, `Shop` checks the availability of the item, nondeterministically assuming the outcome to be either `FAIL` or `OK`. In the former case, the next location is q_0, i.e., the sequence induced by action $\langle \texttt{SellItem}, \texttt{FAIL} \rangle$ ends here. In the second case, ($\langle \texttt{ChkAvail}, \texttt{OK} \rangle$), it proceeds with payment processing in location q_5, again nondeterministically assuming the outcome to be `FAIL` or `OK`. In the former case the next location is q_0, and in the latter case it tries to ship the item (in location q_6), with the expectation of failure.

The underlying signature of $\mathcal{P}_{\texttt{Shop}}$ is $\mathcal{S}_{\texttt{Shop}}$ from Example 3.2. Our $\mathcal{P}_{\texttt{Shop}}$ is a protocol interface because its underlying signature is a signature interface and all its executions can be terminated (by reaching a return location). ∎

Example 3.5 (Concurrency) The following describes the protocol interface for the Store web service:

$$\begin{aligned}
\langle \texttt{ChkStore}, \texttt{OK}\rangle \mapsto \quad & (q_1, \epsilon, q_0), (q_1, \langle \texttt{ChkStore}, \texttt{FAIL}\rangle, q_0), \\
\langle \texttt{ChkStore}, \texttt{FAIL}\rangle \mapsto \quad & (q_2, \langle \texttt{Supp1.GetOffer}, \texttt{REC}\rangle \sqcap \langle \texttt{Supp2.GetOffer}, \texttt{REC}\rangle, q_3), \\
& (q_3, \langle \texttt{Supp1.Order}, \texttt{OK}\rangle, q_0), (q_3, \langle \texttt{Supp2.Order}, \texttt{OK}\rangle, q_0)
\end{aligned}$$

Let us first consider action $\langle \texttt{ChkStore}, \texttt{FAIL}\rangle$. The interface models that two different supplier web services are simultaneously asked to provide an offer. In this case, the protocol interface expresses not only sequence, but also concurrency. After both offers are received, the automaton switches to location q_3, which models that the Store web service orders the missing item from one of the two suppliers. After this, the automaton switches to the return location q_0, i.e., the conversation induced by action $\langle \texttt{ChkStore}, \texttt{FAIL}\rangle$ ends here. The conversation represented by action $\langle \texttt{ChkStore}, \texttt{OK}\rangle$ is either empty, or it contains the actions for ordering new items (for the case the stock is below a certain threshold). This is modeled by making use of the nondeterministic transition relation. ∎

Example 3.6 (Calendar Management Service: Semantics) Let us consider the protocol interface $\mathcal{P}$ of a simple calendar management web service such that $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ where the set of supported actions $\mathcal{J} = \{\langle \texttt{OrganizeMeeting}, \texttt{OK}\rangle, \langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK}\rangle\}$, and the set of required actions $\mathcal{K} = \{\langle \texttt{Busy}, \texttt{OK}\rangle, \langle \texttt{SendConfirmationEmail}, \texttt{OK}\rangle\}$. Then the partial function $\mathcal{R}$ together with the transition relation δ of $\mathcal{H}$ are given as follows,

$$
\begin{array}{ll}
\langle \texttt{OrganizeMeeting}, \texttt{OK} \rangle \mapsto & (q_1, \langle \texttt{SyncCalendars}, \texttt{OK} \rangle, q_2) \\
& (q_2, \langle \texttt{Confirm}, \texttt{OK} \rangle, q_0) \\
\langle \texttt{SyncCalendars}, \texttt{OK} \rangle \mapsto & (q_3, \sqcap \{ \langle \texttt{AddEventToCalendar}_1, \texttt{OK} \rangle, \\
& \langle \texttt{AddEventToCalendar}_2, \texttt{OK} \rangle, \ldots, \\
& \langle \texttt{AddEventToCalendar}_\texttt{k}, \texttt{OK} \rangle \}, q_0) \\
\langle \texttt{Confirm}, \texttt{OK} \rangle \mapsto & (q_4, \langle \texttt{SendConfirmationEmail}, \texttt{OK} \rangle, q_0) \\
& (q_4, \epsilon, q_0) \\
\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK} \rangle \mapsto & (q_5, \epsilon, q_0) \\
\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK} \rangle \mapsto & (q_6, \langle \texttt{Busy}, \texttt{OK} \rangle, q_0)
\end{array}
$$

This protocol interface has the following semantics. The execution begins with the invocation of action $\langle \texttt{OrganizeMeeting}, \texttt{OK} \rangle$, which leads to a number of parallel invocations of $\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK} \rangle$ (one for each desired participant in the meeting being scheduled), sequentially followed by $\langle \texttt{SendConfirmationEmail}, \texttt{OK} \rangle$, which can only be invoked after all parallel invocations of $\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK} \rangle$ have completed their execution. In each invocation of the later action, the system may find out that the participant concerned is busy (leading to the invocation of $\langle \texttt{Busy}, \texttt{OK} \rangle$) or not. After all parallel invocations have finished, the system nondeterministically chooses to send a confirmation email to all participants noting that the meeting has been scheduled, or not: the actual implementation could decide to send the confirmation only if none of the $\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK} \rangle$ actions led to the invocation of $\langle \texttt{Busy}, \texttt{OK} \rangle$, i.e., all desired participants were indeed found to be free.

Now let us consider the following, slightly modified version $\mathcal{P}'$ of the calendar management service:

$$
\begin{array}{ll}
\langle \texttt{OrganizeMeeting}, \texttt{OK}\rangle \mapsto & (q_1, \langle \texttt{SyncCalendars}, \texttt{OK}\rangle, q_2) \\
& (q_2, \langle \texttt{Confirm}, \texttt{OK}\rangle, q_0) \\
\langle \texttt{SyncCalendars}, \texttt{OK}\rangle \mapsto & (q_3, \boxplus\{\langle \texttt{AddEventToCalendar}_1, \texttt{OK}\rangle, \\
& \langle \texttt{AddEventToCalendar}_2, \texttt{OK}\rangle, \ldots, \\
& \langle \texttt{AddEventToCalendar}_\texttt{k}, \texttt{OK}\rangle\}, q_0) \\
\langle \texttt{Confirm}, \texttt{OK}\rangle \mapsto & (q_4, \langle \texttt{SendConfirmationEmail}, \texttt{OK}\rangle, q_0) \\
& (q_4, \epsilon, q_0) \\
\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK}\rangle \mapsto & (q_5, \epsilon, q_0) \\
\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK}\rangle \mapsto & (q_6, \langle \texttt{Busy}, \texttt{OK}\rangle, q_0)
\end{array}
$$

This modified web service does not wait for the execution to complete on all parallel invocations of $\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK}\rangle$. As soon as the first participant's status (busy or not) can be obtained, the system is free to move forward and decide whether to schedule the meeting or not. ∎

3.4.1 Compatibility and Composition

Given two protocol interfaces $\mathcal{P}_1 = (\mathcal{N}_1, \mathcal{J}_1, \mathcal{K}_1, \mathcal{H}_1, \mathcal{R}_1)$ and $\mathcal{P}_2 = (\mathcal{N}_2, \mathcal{J}_2, \mathcal{K}_2, \mathcal{H}_2, \mathcal{R}_2)$, if the underlying signatures $sig(\mathcal{P}_1)$ and $sig(\mathcal{P}_2)$ are compatible, and $Q_1 \cap Q_2 = \emptyset$, then $\mathcal{P}_1$ and $\mathcal{P}_2$ are *compatible* (denoted $comp(\mathcal{P}_1, \mathcal{P}_2)$), and their *composition* (denoted $\mathcal{P}_1 \parallel \mathcal{P}_2$) is $\mathcal{P}_c = (\mathcal{N}_c, \mathcal{J}_c, \mathcal{K}_c, \mathcal{H}_c, \mathcal{R}_c)$, where $\mathcal{N}_c = \mathcal{N}_1 \cup \mathcal{N}_2$, and $\mathcal{J}_c = \mathcal{J}_1 \cup \mathcal{J}_2$, and $\mathcal{K}_c = (\mathcal{K}_1 \cup \mathcal{K}_2) \setminus (\mathcal{N}_c \times \mathcal{I})$, and $\mathcal{H}_c = (Q_1 \cup Q_2, \delta_1 \cup \delta_2)$, and $\mathcal{R}_c = \mathcal{R}_1 \cup \mathcal{R}_2$, where Q_i and δ_i are the set of locations and the transition relation of the automaton $\mathcal{H}_i$ for $i \in \{1, 2\}$. The composition operation is commutative and associative. Compatibility and composition of protocol interfaces can be computed in $O(n^2 \cdot k^2)$ time, where $n = |Q_1| + |Q_2|$ and $k = \max(k_1, k_2)$, and

$k_i = \max_q\{|S| : S = \{(q, term, q') : (q, term, q') \in \delta_i\}\}$ are the corresponding maximal nondeterministic branching factors.

A protocol interface $\mathcal{P}$ is *closed* (*concrete*) if $sig(\mathcal{P})$ is closed (concrete). Given a protocol interface $\mathcal{P}$, an *environment* for $\mathcal{P}$ is a concrete protocol interface $\mathcal{E}$ that is compatible with $\mathcal{P}$, such that the composition $\mathcal{P} \parallel \mathcal{E}$ is closed.

3.4.2 Refinement

Underlying Game Graph

Informally, a protocol interface represents a game being played between two players P_1 and P_2. The interface, together with un-implemented actions in its namespace, collectively constitute the *system*, which plays against the *environment* by making *moves* that change the state of the system and the environment. A *scheduler* (introduced below) arbitrates when both the system and the environment have available moves. Player P_1 plays for the system and the scheduler, and player P_2 plays for the environment. The game is played over a state space consisting of an infinite set of *trees* defined formally as follows.

Given a finite set of *tree symbols* T, a *tree* t over T is a partial function $t : \mathbb{N}^* \to T$, where $\mathbb{N}^*$ denotes the set of finite words over the set of natural numbers $\mathbb{N}$, and the domain $\mathrm{dom}(t) = \{p \in \mathbb{N}^* \mid \exists (p, l) \in t\}$ is prefix-closed. Each element from $\mathrm{dom}(t)$ represents a *node* of tree t: the empty word ρ represents the root of the tree; and the set of child nodes of node p in tree t is $ch(t, p) = \{p' \mid \exists n \in \mathbb{N} : p' = p \cdot n \ \wedge \ p' \in \mathrm{dom}(t)\}$, where $\cdot$ is the concatenation operator for strings over $\mathbb{N}^*$. Each node p of the tree is named with the symbol $t(p)$. The set of leaf nodes of a tree t is $leaf(t) = \{p \in \mathrm{dom}(t) \mid ch(t, p) = \emptyset\}$. The set of all trees over a finite set T is denoted $\mathcal{T}(T)$.

The initial state of the game, and the winning condition will be defined later. The transitions of the game graph are defined as follows.

Intuitively, locations in Q *belong to* the system, and represent control held by an action supported by the interface. Four fresh locations $q^{\forall}$, $q^{\exists}$, q_1, and q_2 are introduced. Location $q^{\forall}$ belonging to the system represents control held by an unimplemented action that will be supported by a refinement of the service being analyzed; the location $q^{\exists}$ belonging to the environment represents control held by an action required by the interface but outside its namespace that will be supported by the environment; locations q_1 and q_2 are return locations.

Informally, the set of transitions $\rightarrow$ of the game graph allows players P_1 and P_2 to change the current state of the game. Players P_1 and P_2 are allowed to move from a given current state s only if s is a pair (t, r) where the tree t has at least one leaf labeled with a location belonging to the system and the environment respectively, and the second component r fulfils the required conditions as described below. Note that if t is a tree with several leaves, not all leaves may necessarily be labeled with locations belonging exclusively to either the system or the environment; then a *scheduler* arbitrates which of them should be allowed to move.[1]If only one of the system or the environment actually has a leaf belonging to it in the tree t, then the scheduler must choose the corresponding player to move next. Otherwise, from a game state (t, ∇), the scheduler chooses either the former or the latter to be allowed to make the next move by choosing a move to $(t, \forall)$ or $(t, \exists)$ respectively. Note that the scheduler is not allowed to modify the tree configuration. The second component r of a game state (t, r) reflects the decision of the scheduler on whether the system or the envi-

[1]Note that the scheduler is a mathematical construct used for theoretical purposes only. We *do not* require web service frameworks *in practice* to incorporate a scheduler to co-ordinate *globally* the computation and communication activities of distributed web services.

ronment is allowed to move next: the system can move only from game states of the form $(t, \forall)$, and the environment can move only from game states of the form $(t, \exists)$. Intuitively, the environment needs to win the game against all possible schedulers. Thus, we allow the scheduler and the system to conspire with each other against the environment, which must play against the coalition. Thus, moves in $\rightarrow$ corresponding to transitions originating from a state (t, r) for $r \in \{\forall, \nabla\}$ belong to player P_1; and those corresponding to transitions originating from a state $(t, \exists)$ belong to player P_2.

Formally, given a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$, the *underlying game graph* of $\mathcal{P}$ is a labeled *two-player game graph* $\mathcal{G} = (S_1, S_2, L, \rightarrow)$ (denoted by $ugs(\mathcal{P})$) where $S_1 = (\mathcal{T}(Q^\bullet) \times \{\forall, \nabla\})$ is the set of *player*-1 *states* at which player P_1 chooses the outgoing transition to the next state, and $S_2 = (\mathcal{T}(Q^\bullet) \times \{\exists\})$ is the set of *player*-2 *states* where player P_2 chooses the next state, and $L = 2^{\mathcal{A} \cup \{ret\}}$ is the set of *transition labels*, and $\rightarrow \subseteq S \times L \times S$ is the *game transition relation*; where the set of *game states* $S = S_1 \cup S_2$ is the set of pairs (t, r) with r being an element of the set $R = \{\exists, \forall, \nabla\}$ and t being a tree over the set of tree symbols $Q^\bullet = \{(q, \bullet) \mid q \in Q \cup \{q^\forall, q^\exists, q_1, q_2\}, \bullet \in \{\boxplus, \circ\}\}$, with Q is the set of locations of protocol automaton $\mathcal{H}$ and $q^\forall, q^\exists, q_1, q_2$ are fresh symbols not in Q, and the transitions between states are labeled with sets of elements from $\mathcal{A} \cup \{ret\}$. We write $(t, r) \xrightarrow{A} (t', r')$ for $((t, r), A, (t', r')) \in \rightarrow$. Given a protocol interface $\mathcal{P}$ the corresponding game transition relation is defined as follows. In the following, for an action a the symbol q_a is defined as follows: $q_a = \mathcal{R}(a)$ if a is supported ($a \in \mathcal{J}$); $q_a = q^\forall$ if $a \notin \mathcal{J}$ but $[a] \in \mathcal{N}$; and $q_a = q^\exists$ if $[a] \notin \mathcal{N}$. The relation $\delta^\bullet$ used below is defined as $\delta^\bullet = \delta \cup \delta^\forall \cup \delta^\exists$, where δ is the transition relation

of $\mathcal{H}$, and

$$\begin{aligned}
\delta^{\forall} &= \{(q^{\forall}, \epsilon, q_1)\} \cup \\
&\quad \{(q^{\forall}, a, q^{\forall}) \mid a \in \mathcal{A}, \text{ such that } [a] \in \mathcal{N} \text{ or } a \in \mathcal{K}\} \cup \\
&\quad \{(q^{\forall}, \circ A, q^{\forall}) \mid \circ \in \{\sqcap, \boxplus\}, A \subseteq \mathcal{A}, \text{ s.t. for all } a \in A, [a] \in \mathcal{N} \text{ or } a \in \mathcal{K}\}, \text{and} \\
\delta^{\exists} &= \{(q^{\exists}, \epsilon, q_2)\} \cup \\
&\quad \{(q^{\exists}, a, q^{\exists}) \mid a \in \mathcal{A}\} \cup \\
&\quad \{(q^{\exists}, \circ A, q^{\exists}) \mid \circ \in \{\sqcap, \boxplus\}, A \subseteq \mathcal{A}\}.
\end{aligned}$$

Informally, the relation $\delta^{\forall}$ encodes the ability of an unimplemented action in the interface's namespace to invoke any action under the restrictions imposed on it by the interface; and the relation $\delta^{\exists}$ encodes the ability of the environment to invoke any action. The relation $\rightarrow$ is defined as follows:

- Next-Mover: $(t, \nabla) \xrightarrow{\epsilon} (t, r)$ where $r = \forall$ (or $\exists$) if there exists a node p such that $p \in \mathit{leaf}(t)$, and $t(p) = (q, \circ)$, and $q \in Q \cup \{q^{\forall}, q_1\}$ (or $q \in \{q^{\exists}, q_2\}$).

- Epsilon: $(t, r) \xrightarrow{\epsilon} (t', \nabla)$ if there exists a node p such that $p \in \mathit{leaf}(t)$, and $t(p) = (q, \circ)$, and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^{\forall}, q_1\}$ (or $q \in \{q^{\exists}, q_2\}$), and $(q, \epsilon, q') \in \delta^{\bullet}$, and $t' = (t \setminus \{(p, (q, \circ))\}) \cup \{(p, (q', \circ))\}$.

- Call: $(t, r) \xrightarrow{\{a\}} (t', \nabla)$ if there exists a node p such that $p \in \mathit{leaf}(t)$, and $t(p) = (q, \circ)$, and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^{\forall}, q_1\}$ (or $q \in \{q^{\exists}, q_2\}$), and $(q, a, q') \in \delta^{\bullet}$, and $t' = (t \setminus \{(p, (q, \circ))\}) \cup \{(p, (q', \circ)), (p \cdot 0, (q_a, \circ))\}$.

- Fork: $(t, r) \xrightarrow{A} (t', \nabla)$ if there exists a node p such that $p \in \mathit{leaf}(t)$, and $t(p) = (q, \circ)$, and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^{\forall}, q_1\}$ (or $q \in \{q^{\exists}, q_2\}$), and $(q, \sqcap A, q') \in \delta^{\bullet}$, and $t' = (t \setminus \{(p, (q, \circ))\}) \cup \{(p, (q', \circ)), (p \cdot 0, (q_{a_0}, \circ)), \ldots, (p \cdot k, (q_{a_k}, \circ))\}$, where $A = \{a_0, \ldots, a_k\}$.

- Fork-Choice: $(t, r) \xrightarrow{A} (t', \nabla)$ if there exists a node p such that $p \in \mathit{leaf}(t)$, and $t(p) = (q, \circ)$, and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^{\forall}, q_1\}$ (or $q \in \{q^{\exists}, q_2\}$), and $(q, \boxplus A, q') \in \delta^{\bullet}$, and $t' = (t \setminus \{(p, (q, \circ))\}) \cup \{(p, (q', \boxplus)), (p \cdot 0, (q_{a_0}, \circ)), \ldots, (p \cdot k, (q_{a_k}, \circ))\}$, where $A = \{a_0, \ldots, a_k\}$.

- Return: $(t, r) \xrightarrow{\{ret\}} (t', \nabla)$ if there exists a node $p \cdot n$, $n \in \mathbb{N}$, such that $p \cdot n \in \mathit{leaf}(t)$, and $t(p \cdot n) = (q, \circ)$ and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^{\forall}, q_1\}$ (or $q \in \{q^{\exists}, q_2\}$), such that q is a return location, $t(p) = (q', \circ)$, and $t' = t \setminus \{(p \cdot n, (q, \circ))\}$.

- Return & Remove Sibling Tree: $(t, r) \xrightarrow{\{ret\}} (t', \nabla)$ if there exists a node $p \cdot n$, $n \in \mathbb{N}$, such that $p \cdot n \in \mathit{leaf}(t)$, and $t(p \cdot n) = (q, \circ)$ and $r = \forall$ (or $\exists$) if $q \in Q \cup \{q^{\forall}, q_1\}$ (or $q \in \{q^{\exists}, q_2\}$), such that q is a return location, $t(p) = (q'', \boxplus)$, and $t' = (t \setminus \{(p \cdot p', (q', \bullet)) \mid p' \in \mathbb{N}^* \wedge (q', \bullet) \in Q^{\bullet}\}) \cup \{(p, (q'', \circ))\}$.

A *run* of the game structure is an alternating sequence of game states and sets of actions $s_0, A_1, s_1, A_2, s_2, \ldots$, with $\forall i \in \{1, \ldots n\} : s_{i-1} \xrightarrow{A_i} s_i$. A *trace* is the projection of a run to its action sets; for a run $s_0, A_1, s_1, A_2, s_2, \ldots$, the corresponding trace is $A_1, A_2, \ldots$. The set of *moves* belonging to player P_i at a game state $s_j \in S_i$ is $\{s \mid s_j \xrightarrow{A} s, A \subseteq \mathcal{A} \cup \{ret\}\}$, where $i \in \{1, 2\}$. Move s of player P_i for $i \in \{1, 2\}$ at game state $s_j \in S_i$ *is labeled with* A if $s_j \xrightarrow{A} s$. A *strategy* σ_i of a player P_i for $i \in \{1, 2\}$ is a function that maps every finite run $s_0, A_1, s_1, A_2, s_2, \ldots, s_k$ such that $s_k \in S_i$ to a move available to P_i at s_k. The set of strategies of player P_i for $i \in \{1, 2\}$ is denoted Ψ_i. For a location q, a *q-run* is a run $s_0, A_1, s_1, \ldots$ with $s_0 = (\{(\rho, (q, \circ))\}, \nabla)$, i.e., a run starting from a game state where the sole thread of control rests at location q of the protocol automaton; a *q-trace* is a trace corresponding to a q-run. For a given pair of strategies (σ_1, σ_2) and a location q, the *outcome* is a q-run $s_0, A_1, s_1, A_2, s_2, \ldots$, such that for every prefix $r_i = s_0, A_1, s_1, A_2, \ldots, s_i$ of the

run, if $s_i \in S_j$, we have $\sigma_j(r_i) = s_{i+1}$, for $i \in \mathbb{N}$ and $j \in \{1, 2\}$. Given a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ and an action $a \in \mathcal{J}$, the *initial state of the game representing the invocation of a on* $\mathcal{P}$ (denoted $init(\mathcal{P}, a)$) is $(\{\rho \mapsto (\mathcal{R}(a), \circ)\}, \nabla)$.

Alternating Simulation

Given two two-player game graphs $\mathcal{G}' = (S'_1, S'_2, L, \rightarrow')$ and $\mathcal{G} = (S_1, S_2, L, \rightarrow)$ with state spaces $S' = S'_1 \cup S'_2$ and $S = S_1 \cup S_2$ respectively, we say $\mathcal{G}'$ *is in alternating simulation with* $\mathcal{G}$ if there exists a relation $\precsim \subseteq S' \times S$ such that:

i) for every $s_1 \in S_1$, and $s'_1 \in S'_1$, if $s'_1 \precsim s_1$, then for every player P_1 move s'_2 labeled with A and available at s'_1 there exists a player P_1 move s_2 labeled with A and available at s_1, such that $s'_2 \precsim s_2$, and

ii) for every $s_2 \in S_2$, and $s'_2 \in S'_2$, if $s'_2 \precsim s_2$, then for every player P_2 move s_1 labeled with A and available at s_2, there exists a player P_2 move s'_1 labeled with A and available at s'_2, such that $s'_1 \precsim s_1$.

Given two protocol interfaces $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ and $\mathcal{P}' = (\mathcal{N}', \mathcal{J}', \mathcal{K}', \mathcal{H}', \mathcal{R}')$, we say $\mathcal{P}'$ *refines* $\mathcal{P}$ (written $\mathcal{P}' \preccurlyeq \mathcal{P}$), if:

i) $sig(\mathcal{P}') \preccurlyeq sig(\mathcal{P})$, and

ii) for every action $a \in \mathcal{A}$, if $\mathcal{P}$ supports a, then the two two-player game graphs $\mathcal{G}' = ugs(\mathcal{P}')$ and $\mathcal{G} = ugs(\mathcal{P})$ are such that there exists an alternating simulation relation $\precsim$ with $init(\mathcal{P}', a) \precsim init(\mathcal{P}, a)$.

Theorem 3.3 *Let $\mathcal{P}_1$, $\mathcal{P}'_1$ and $\mathcal{P}_2$ be three protocol interfaces such that $\mathcal{P}'_1 \preccurlyeq \mathcal{P}_1$ and $comp(\mathcal{P}_1, \mathcal{P}_2)$. Then $comp(\mathcal{P}'_1, \mathcal{P}_2)$ and $(\mathcal{P}'_1 \parallel \mathcal{P}_2) \preccurlyeq (\mathcal{P}_1 \parallel \mathcal{P}_2)$.*

Proof.

i) Given $\mathcal{P}_1' \preccurlyeq \mathcal{P}_1$, we get, by definition of refinement for protocol interfaces, $sig(\mathcal{P}_1') \preccurlyeq sig(\mathcal{P}_1)$. Then, by definition of refinement for signature interfaces, $\mathcal{N}_1' \subseteq \mathcal{N}_1$, where $\mathcal{N}_1'$ and $\mathcal{N}_1$ are the namespaces of signature interfaces $sig(\mathcal{P}_1')$ and $sig(\mathcal{P}_1)$ respectively.

Now, given $comp(\mathcal{P}_1, \mathcal{P}_2)$, we have, by definition of compatibility of protocol interfaces, $comp(sig(\mathcal{P}_1), sig(\mathcal{P}_2))$. Then, by definition of compatibility for signature interfaces, we have $\mathcal{N}_1 \cap \mathcal{N}_2 = \emptyset$, where $\mathcal{N}_1$ and $\mathcal{N}_2$ are the namespaces of the signature interfaces $sig(\mathcal{P}_1)$ and $sig(\mathcal{P}_2)$ respectively.

From the above, we have $\mathcal{N}_1' \cap \mathcal{N}_2 = \emptyset$, and thence, we get $comp(sig(\mathcal{P}_1'), sig(\mathcal{P}_2))$, and thus, we conclude $comp(\mathcal{P}_1', \mathcal{P}_2)$.

ii) Let us assume that the required result does not hold. In other words, let $\mathcal{P}_1' \preccurlyeq \mathcal{P}_1$, but $(\mathcal{P}_1' \parallel \mathcal{P}_2) \not\preccurlyeq (\mathcal{P}_1 \parallel \mathcal{P}_2)$. Note that by definitions of composition for protocol interfaces, underlying signatures for protocol interfaces, and refinement for signature interfaces, we have $sig(\mathcal{P}_1' \parallel \mathcal{P}_2) \preccurlyeq sig(\mathcal{P}_1 \parallel \mathcal{P}_2)$. Thus, by definition of refinement for protocol interfaces, and by our assumption $(\mathcal{P}_1' \parallel \mathcal{P}_2) \not\preccurlyeq (\mathcal{P}_1 \parallel \mathcal{P}_2)$, we conclude that there must be at least one action $a \in \mathcal{J}_1 \cup \mathcal{J}_2$, where $\mathcal{P}_i = (\mathcal{N}_i, \mathcal{J}_i, \mathcal{K}_i, \mathcal{H}_i, \mathcal{R}_i)$ for $i \in \{1, 2\}$ and $\mathcal{P}_1' = (\mathcal{N}_1', \mathcal{J}_1', \mathcal{K}_1', \mathcal{H}_1', \mathcal{R}_1')$, such that $ugs(\mathcal{P}_1' \parallel \mathcal{P}_2, a)$ is not in alternating simulation with $ugs(\mathcal{P}_1 \parallel \mathcal{P}_2, a)$. To make this happen, the only way is for an action supported by $\mathcal{P}_1'$ but not by $\mathcal{P}_1$ to invoke additional actions. However, since $\mathcal{K}_1' \subseteq \mathcal{K}_1$ by definition of refinement for protocol interfaces (and for signature interfaces), and by the definition of the underlying game graphs for protocol interfaces as defined above,

we see that this is not possible. Thus, a contradiction is reached, and the required result follows. ∎

3.4.3 Specifications

A *conversation* of a protocol interface is a set of *sequences* of objects A, where each A is a set of actions. A *specification* ψ for a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ is a temporal safety property of the form $\psi = a \not\rightsquigarrow \varphi$, where φ is a temporal-logic formula of the following form:

$$\varphi ::= \phi \wedge (\phi \,\mathcal{U}\, \varphi) \mid \phi$$
$$\phi ::= \textsc{t} \mid \textsc{f} \mid b \mid \neg b \mid \phi \wedge \phi \mid \phi \vee \phi$$

where $a \in \mathcal{J}$ and $b \in \mathcal{A}$.

The temporal-logic formula $\phi \,\mathcal{U}\, \varphi$ (read "ϕ until φ") represents a temporal property of conversations. Intuitively, a conversation satisfies a formula $\phi \,\mathcal{U}\, \varphi$ if it satisfies φ eventually, and satisfies ϕ at each step until then. A specification $a \not\rightsquigarrow \varphi$ means that the invocation of action a must not lead to a conversation satisfying φ. A specification ψ for a protocol interface $\mathcal{P}$ is interpreted over traces generated by the *underlying game graph* of $\mathcal{P}$. Essentially, the specification is taken as the winning condition for the game.

Formally, given a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$, a location q of $\mathcal{H}$ *satisfies* a temporal formula φ (written $q \models \varphi$) if for all strategies $\sigma_2 \in \Psi_2$ of player P_2, there exists a strategy $\sigma_1 \in \Psi_1$ of player P_1, such that the outcome is a run corresponding to a q-trace $\pi = A_1, A_2, \ldots$ of the underlying game graph $ugs(\mathcal{P})$ such that $\pi \models_t \varphi$, where the satisfaction relation $\models_t$ between traces and temporal-logic formulae is defined as follows. For the trace $\sigma = A_1, A_2, \ldots, A_k$, we have $\sigma \models_t a$

if $a \in A_1$ and $\sigma \models_t \neg a$ otherwise; and $\sigma \models_t \phi_1 \wedge \phi_2$ if $\sigma \models_t \phi_1$ and $\sigma \models_t \phi_2$; and $\sigma \models_t \phi_1 \vee \phi_2$ if $\sigma \models_t \phi_1$ or $\sigma \models_t \phi_2$; and $\sigma \models_t \mathrm{T}$ for all σ; and $\sigma \models_t \mathrm{F}$ for no σ. For a trace $\sigma = A_1, A_2, \ldots, A_k$, we have $\sigma \models_t \phi_1 \,\mathcal{U}\, (\phi_2 \wedge (\phi_3 \,\mathcal{U}\, \varphi))$ if either (1) $\bigwedge A_1 \wedge (\neg \bigvee(\mathcal{A} \setminus A_1)) \Rightarrow \phi_1$ and $\sigma' \models_t \phi_2$ and $\sigma' \models_t \phi_3 \,\mathcal{U}\, \varphi$ where $\sigma' = A_2, A_3, \ldots, A_k$, or (2) $\sigma \models_t \phi_2$ and $\sigma \models_t \phi_3 \,\mathcal{U}\, \varphi$. A protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ satisfies a specification $a \not\rightsquigarrow \varphi$ (written $\mathcal{P} \models \psi$) if we have $q \not\models \varphi$ where $q = \mathcal{R}(a)$,

Theorem 3.4 *Given a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ and a specification $\psi = a \not\rightsquigarrow \varphi$, we have $\mathcal{P} \models \psi$ if and only if for all concrete protocol interfaces $\mathcal{P}' = (\mathcal{N}', \mathcal{J}', \mathcal{K}', \mathcal{H}', \mathcal{R}')$ such that $\mathcal{P}' \preccurlyeq \mathcal{P}$, there exists an environment $\mathcal{E}_{\mathcal{P}'}$ of $\mathcal{P}'$, such that $(\mathcal{P}' \parallel \mathcal{E}_{\mathcal{P}'}) \models \psi$.*

Proof.

($\Rightarrow$) Let us assume that the required result does not hold. Then, we have, given $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$ and a specification $\psi = a \not\rightsquigarrow \varphi$, and that $\mathcal{P} \models \psi$, and by assumption, that there exists a protocol interface $\mathcal{P}' = (\mathcal{N}', \mathcal{J}', \mathcal{K}', \mathcal{H}', \mathcal{R}')$ such that $\mathcal{P}' \preccurlyeq \mathcal{P}$, but for all environments $\mathcal{E}_{\mathcal{P}'}$ of $\mathcal{P}'$, $(\mathcal{P}' \parallel \mathcal{E}_{\mathcal{P}'}) \not\models \psi$. Now, we have three cases:

a) a is outside namespace $\mathcal{N}$: Then a belongs to the namespace of $\mathcal{E}_{\mathcal{P}'}$, and hence the assumption cannot occur. A contradiction is reached, and hence this subcase has to be rejected if our assumption has to be true.

b) a is in namespace $\mathcal{N}$, but $a \notin \mathcal{J}$: Then by definition of the underlying game graph of $\mathcal{P}$, we see that for every tuple $(q_a, term, q) \in \delta'$, we must have $(q^{\forall}, term, q^{\forall}) \in \delta^{\bullet}$,where $\delta^{\bullet} = \delta \cup \delta^{\forall} \cup \delta^{\exists}$ as defined before, and δ' and δ are the transition relations of the protocol automata $\mathcal{H}'$ and $\mathcal{H}$ respectively.

Hence, if $(\mathcal{P}' \parallel \mathcal{E}_{\mathcal{P}'}) \not\models \psi$, then a must generate a trace in $(\mathcal{P}' \parallel \mathcal{E}_{\mathcal{P}'})$ that violates the temporal formula φ. However, then by our above conclusion, the same trace can be generated by a in $\mathcal{P}$, and thus, we have $\mathcal{P} \not\models \psi$, thus reaching a contradiction.

c) $a \in \mathcal{J}$: In this subcase, by definition of refinement for protocol interfaces, we directly have that $ugs(\mathcal{P}', a) \precsim ugs(\mathcal{P}, a)$, which rules out the possibility of a generating a trace in $\mathcal{P}'$ that violates the temporal formula φ, since we know that no such trace can be generated by a in $\mathcal{P}$, given $\mathcal{P} \models \psi$. Thus, the assumption is not possible in this subcase either.

Since all possible subcases were considered, our assumption must be false, and hence the required result follows.

($\Leftarrow$) Given that for all protocol interfaces $\mathcal{P}'$ such that $\mathcal{P}' \preccurlyeq \mathcal{P}$, there exist environments $\mathcal{E}_{\mathcal{P}'}$ of $\mathcal{P}'$ such that $(\mathcal{P}' \parallel \mathcal{E}_{\mathcal{P}'}) \models \psi$, we reach, by reflexivity of the refinement relation for protocol interfaces, that $\mathcal{P}$ has an environment $\mathcal{E}_{\mathcal{P}}$ such that $(\mathcal{P} \parallel \mathcal{E}_{\mathcal{P}}) \models \psi$. Now, from that, and from the definition of the underlying game graph of $\mathcal{P}$, we conclude that in that game graph there exist choices of $\delta^{\exists}$ for which every q_a-trace generated does not satisfy the temporal formula φ. From this, by definition of specification satisfactio for protocol interfaces, we conclude $\mathcal{P} \models \psi$. ∎

Corollary 3.2 *Let $\mathcal{P}_1$, $\mathcal{P}_1'$, and $\mathcal{P}_2$ be three protocol interfaces such that $\mathcal{P}_1' \preccurlyeq \mathcal{P}_1$ and $comp(\mathcal{P}_1, \mathcal{P}_2)$. Let ψ be a specification such that $(\mathcal{P}_1 \parallel \mathcal{P}_2) \models \psi$. Then $(\mathcal{P}_1' \parallel \mathcal{P}_2) \models \psi$.*

While the formalism in [25] allowed specification-checking only for closed protocol interfaces, the formalism as presented in this dissertation is based on [26] which

allows checking specifications for open protocol interfaces as well. The specification language as presented in this dissertation is based on that from [27], which is an enhanced version of the specification language used in [25, 26]. We are able to check specifications in this language using Algorithm 2 which uses the set of proof rules presented in Figures 3.2 through 3.14, where q_a for an action $a \in \mathcal{A}$ is defined as in Section 3.4.2. The rules are written taking into account the temporal logic for specifications, the interleaving semantics of executions by parallel subtrees of the overall system configuration at each stage, and the non-completing semantics of the operator $\boxplus$. For ease of presentation, we use the following abbreviations in the proof rules. Let $\mathbb{N}_{2j+1,2k+1}$ with $j \leq k$ be the set of naturals $\{2j+1, 2j+1, \ldots, 2k, 2k+1\}$. Given a set of naturals $M = \mathbb{N}_{2j+1,2k+2} \setminus \{2i \mid i \in N\}$ where N is some subset of $\mathbb{N}$ such that $2k+2 \notin N$, then $\varphi^0_{2j+1,2k+2}$ is an abbreviation for the formula $\phi_{2j+1}\, \mathcal{U}\, (\phi_{2j+2} \wedge (\phi_{2j+3}\, \mathcal{U}\, (\phi_{2j+4} \wedge \ldots\, \mathcal{U}\, (\phi_{2k} \wedge (\phi_{2k+1}\, \mathcal{U}\, \phi_{2k+1})))))$ where $\phi_{2i} = \mathrm{T}$ for every $i \in N$; and $\mathbb{N}(\varphi^0_{2j+1,2k+2})$ denotes the set M. Given a set of naturals $M = \mathbb{N}_{2j+1,2k+1} \setminus \{2i \mid i \in N\}$ where N is some subset of $\mathbb{N}$, then $\varphi_{2j+1,2k+1}$ is an abbreviation for the formula $\phi_{2j+1}\, \mathcal{U}\, (\phi_{2j+2} \wedge (\phi_{2j+3}\, \mathcal{U}\, (\phi_{2j+4} \wedge \ldots\, \mathcal{U}\, (\phi_{2k} \wedge \Box\phi_{2k+1}))))$ where $\phi_{2i} = \mathrm{T}$ for every $i \in N$; and $\mathbb{N}(\varphi_{2j+1,2k+1})$ denotes the set M. Given a set of naturals $M = \mathbb{N}_{2j+1,2k+1} \setminus \{2i \mid i \in N\}$ where N is some subset of $\mathbb{N}$, then $\varphi'_{2j+1,2k+1}$ is an abbreviation for the formula $\phi_{2j+1}\, \mathcal{U}\, (\phi_{2j+2} \wedge (\phi_{2j+3}\, \mathcal{U}\, (\phi_{2j+4} \wedge \ldots\, \mathcal{U}\, (\phi_{2k} \wedge (\phi_{2k+1}\, \mathcal{U}\, \mathrm{T})))))$ where $\phi_{2i} = \mathrm{T}$ for every $i \in N$; and $\mathbb{N}(\varphi'_{2j+1,2k+1})$ denotes the set M. The formula $(\phi_{2j_0+1}\, \mathcal{U}\, (\phi_{2j_0+2} \wedge \ldots \wedge\, \mathcal{U}\, (\phi_{2j_3} \wedge \Box\phi_{2j_3+1})))$ is defined to be $\Box\phi_{2j_3+1}$ when $j_0 = j_3$.

We demonstrate the practical usefulness of this specification formalism in Example 3.7 and in Section 3.5.

Example 3.7 (Calendar Management Service: Specification Check-

ing) Let us consider the calendar management protocol interface $\mathcal{P}$ as defined in Example 3.6, and the specification $\psi = \langle \texttt{OrganizeMeeting}, \texttt{OK} \rangle \not\rightsquigarrow \textsc{t}\ \mathcal{U}\ (\langle \texttt{SendConfirmationEmail}, \texttt{OK} \rangle \wedge (\textsc{t}\ \mathcal{U}\ \langle \texttt{Busy}, \texttt{OK} \rangle))$. Intuitively, the specification ψ represents the question "Is there an execution trace resulting from the invocation of action $\langle \texttt{OrganizeMeeting}, \texttt{OK} \rangle$ on which at least one desired participant is found to be busy after the email confirmation for the meeting had already been sent out?" As expected, we find that $\mathcal{P} \models \psi$, and $\mathcal{P}' \not\models \psi$. ∎

Proposition 3.1 (Correctness of specification checking) *For a given protocol interface $\mathcal{P}$ and a specification $\psi = a \not\rightsquigarrow \varphi$ for $\mathcal{P}$, procedure* $\mathsf{CheckSpec}(\mathcal{P}, a, \varphi)$ *(Algorithm 2) terminates and returns* Yes *if $\mathcal{P}$ satisfies ψ, and* No *otherwise.*

Proof (sketch).

1: $\mathcal{P}$ satisfies the specification ψ: Then, the only way the Algorithm can return a wrong answer (No) is if it manages to prove at least one temporal formula that is not in fact fulfilled in the interface. Then, at least one of the proof rules must be unsound. However, from the discussion accompanying the proof rules we observe that every proof rule presented is sound, i.e.the conclusion of each rule is satisfied if the premises hold. Therefore it is not possible for the algorithm to return the answer No incorrectly.

2: $\mathcal{P}$ does not satisfy the specification ψ: Then, the only way the Algorithm can return a wrong answer (YES) is it is fails to prove at least one temporal logic formula that is in fact fulfilled by the interface. However, from the discussion accompanying the proof rules we observe that the conclusion of each rule is satisfied only if the premises hold. i.e. the proof rules are complete. It is

$$\frac{}{q \models \mathrm{T}} \qquad (\mathrm{T}^1)$$

$$\frac{}{q \models \phi \,\mathcal{U}\, \mathrm{T}} \qquad (\mathrm{T}^2)$$

$$\frac{}{q \models a} \quad (q, a, q') \in \delta \cup \delta^{\forall} \ \vee \ ((q, \circ A, q') \in \delta \ \wedge \ a \in A), \ \circ \in \{\sqcap, \boxplus\}. \qquad (a)$$

$$\frac{}{q \models \neg a} \quad (q, a, q') \text{ and/or } (q, \circ A, q'') \text{ (such that } a \in A) \text{ is/are not the only one/two tuples of the form } (q, ?, ?) \text{ in } \delta, \circ \in \{\sqcap, \boxplus\}. \qquad (\neg a)$$

$$\frac{q \models \phi_1 \quad q \models \phi_2 \,\mathcal{U}\, \varphi}{q \models \phi_1 \wedge (\phi_2 \,\mathcal{U}\, \varphi)} \qquad (\wedge^1)$$

$$\frac{q \models \phi_1 \quad q \models \phi_2}{q \models \phi_1 \wedge \phi_2} \qquad (\wedge^2)$$

$$\frac{q \models \phi_1}{q \models \phi_1 \vee \phi_2} \qquad (\vee)$$

$$\frac{q' \models \varphi}{q \models \varphi} \quad (q, \epsilon, q') \in \delta. \qquad (\epsilon\, \mathcal{U})$$

Figure 3.2. Proof rules for specification checking (part 1)

$$\frac{}{q \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge (\phi_3 \,\mathcal{U}\, (\phi_4 \wedge \ldots \mathcal{U}\, \phi_{2k})))}$$

$$\begin{array}{l}((q,c,q') \in \delta \cup \delta^{\forall} \ \wedge \\ (c \ \wedge \ \neg \bigvee(\mathcal{A} \setminus \{c\})) \Rightarrow \bigwedge_{1 \le i \le k} \phi_{2i}) \ \vee \\ ((q, \circ A, q') \in \delta \ \wedge \\ (\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow \bigwedge_{1 \le i \le k} \phi_{2i}), \\ \circ \in \{\sqcap, \boxplus\}.\end{array}$$

(Reached $\mathcal{U}^0$)

$$\frac{q_c \models (\phi_{2j_0+1} \,\mathcal{U}\, (\phi_{2j_0+2} \wedge \ldots \wedge \mathcal{U}\, (\phi_{2k})))}{q \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge (\phi_3 \,\mathcal{U}\, (\phi_4 \wedge \ldots \mathcal{U}\, \phi_{2k})))}$$

$$\begin{array}{l}(q,c,q') \in \delta, \\ c \ \wedge \ \neg \bigvee(\mathcal{A} \setminus \{c\}) \Rightarrow \\ (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \\ 0 \le j_0 < k.\end{array}$$

(Reached $\mathcal{U}_1^+$)

$$\frac{\begin{array}{l}q_{a_1} \models \varphi'_{2j_{11}+1,2j_{12}+1} \\ q_{a_2} \models \varphi'_{2j_{21}+1,2j_{22}+1} \\ \ldots \\ q_{a_l} \models \varphi^0_{2j_{l1}+1,2j_{l2}+2} \\ \ldots \\ q_{a_k} \models \varphi'_{2j_{k1}+1,2j_{k2}+1}\end{array}}{q \models \varphi^0_{1,2j_1+2}}$$

$$\begin{array}{l}(q, \circ A, q') \in \delta, A = \{a_1, a_2, \ldots, a_l, \ldots, a_k\}, \\ \bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \\ \circ \in \{\sqcap, \boxplus\}, j_0 \le j_{i1} \le j_{i2} \le j_1 \text{ for all } 1 \le i \le k, \\ \text{such that } \mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1}) \cap \mathbb{N}(\varphi'_{2j_{m1}+1,2j_{m2}+1}) \\ \text{contains no even natural for all } 1 \le i \le k, \\ 1 \le m \le k, i \ne m, i \ne l, m \ne l, \text{ and} \\ \mathbb{N}(\varphi^0_{2j_{l1}+1,2j_{l2}+2}) \cap \mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1}) \text{ contains no} \\ \text{even natural for all } 1 \le i \le k, i \ne l, \text{ and} \\ \bigcup_{1 \le i \le k}^{i \ne l} \mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1}) \cup \mathbb{N}(\varphi^0_{2j_{l1}+1,2j_{l2}+2}) = \\ \mathbb{N}_{2j_0+1,2j_1+2}, \text{ (and } j_{l2} = j_1).\end{array}$$

(Reached $\mathcal{U}_2^+$)

Figure 3.3. Proof rules for specification checking (part 2)

$$\frac{q_a \models (\phi_{2j_0+1}\,\mathcal{U}\,(\phi_{2j_0+2} \wedge \ldots \wedge \mathcal{U}\,(\phi_{2j_3} \wedge \Box\phi_{2j_3+1}))) \quad q' \models \phi_{2j_3+1}\,\mathcal{U}\ldots\mathcal{U}\,\phi_{2k}}{q \models \phi_1\,\mathcal{U}\,(\phi_2 \wedge (\phi_3\,\mathcal{U}\,(\phi_4 \wedge \ldots\,\mathcal{U}\,\phi_{2k})))}$$

$$a \;\wedge\; \neg \bigvee(\mathcal{A} \setminus a) \Rightarrow (\textstyle\bigwedge_{0\le i\le j_0} \phi_{2i}) \wedge \phi_{2j_0+1},\ (q,a,q') \in \delta, 0 \le j_0 \le j_3 < k.$$

(Call $\mathcal{U}$)

$$\frac{\begin{array}{l} q' \models \phi_{2j_0+1}\,\mathcal{U}\ldots\mathcal{U}\,\phi_{2k} \\ q' \models \phi_{2j_0+3}\,\mathcal{U}\ldots\mathcal{U}\,\phi_{2k} \\ q' \models \phi_{2j_0+5}\,\mathcal{U}\ldots\mathcal{U}\,\phi_{2k} \\ \ldots \\ q' \models \phi_{2k-3}\,\mathcal{U}\,(\phi_{2k-2} \wedge (\phi_{2k-1}\,\mathcal{U}\,\phi_{2k})) \\ q' \models \phi_{2k-1}\,\mathcal{U}\,\phi_{2k} \end{array}}{q \models \phi_1\,\mathcal{U}\,(\phi_2 \wedge (\phi_3\,\mathcal{U}\,(\phi_4 \wedge \ldots\,\mathcal{U}\,\phi_{2k})))}$$

$$\begin{array}{l} a \;\wedge\; \neg \bigvee(\mathcal{A} \setminus a) \Rightarrow \\ (\textstyle\bigwedge_{0\le i\le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \\ (q,a,q') \in \delta, q_a = q^{\exists}, \\ \neg\phi_{2j_0+1} \implies \phi_{2j_0+2}, \\ \neg\phi_{2j_0+3} \implies \phi_{2j_0+4}, \\ \ldots, \\ \neg\phi_{2k-3} \implies \phi_{2k-2}. \\ \neg\phi_{2k-1} \implies \phi_{2k}. \end{array}$$

(Call $\mathcal{U}$ $\exists$)

Figure 3.4. Proof rules for specification checking (part 3)

$$\frac{\begin{array}{l} q_{a_1} \models \varphi_{2j_{11}+1,2j_{12}+1} \\ q_{a_2} \models \varphi_{2j_{21}+1,2j_{22}+1} \\ \cdots \\ q_{a_k} \models \varphi_{2j_{k1}+1,2j_{k2}+1} \\ q' \models \varphi^0_{2j_1+1,2j_2+2} \end{array}}{q \models \varphi^0_{1,2j_2+2}}$$

$$\begin{array}{l} (\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \\ (q, \sqcap A, q') \in \delta, A = \{a_1, a_2, \ldots, a_k\}, \\ j_0 \le j_{i1} \le j_{i2} \le j_1 \text{ for all } 1 \le i \le k, \text{ such that} \\ \mathbb{N}(\varphi_{2j_{i1}+1,2j_{i2}+1}) \cap \mathbb{N}(\varphi_{2j_{m1}+1,2j_{m2}+1}) \text{ contains no} \\ \text{even natural for all } 1 \le i \le k, 1 \le m \le k, i \ne m, \\ \text{and } \bigcup_{1 \le i \le k} \mathbb{N}(\varphi_{2j_{i1}+1,2j_{i2}+1}) = \mathbb{N}_{2j_0+1,2j_1+1}. \end{array}$$

(Fork $\mathcal{U}$)

Figure 3.5. Proof rules for specification checking (part 4)

therefore impossible for the algorithm to terminate failing to prove a temporal logic formula necessary to prove in order to check satisfaction of the given ψ. Thus it is not possible for the algorithm to return the answer YES incorrectly. From this and the above paragraph, the required result follows. ∎

Note that we check specifications for open protocol interfaces, which contain calls to unsupported actions, either in its own namespace, corresponding to actions that are going to be supported in refined versions, or in the environment's namespace, corresponding to actions that will be supported only when the interface is composed with the environment and additional functionality becomes available. In our case, the game graph allows a variety of choices for behavior for unsupported environment or refinement actions, and the actual behavior of the environment and of the refinement is found by solving the game. In our framework, the environment is friendly, and its goal while playing the game is to help the interface satisfy the specification. The behavior of the system is constrained by the given code, and it has behavioral free-

$$\frac{\begin{array}{l} q_{a_{11}} \models \varphi_{2j_{111}+1,2j_{112}+1} \\ q_{a_{12}} \models \varphi_{2j_{121}+1,2j_{122}+1} \\ \ldots \\ q_{a_{1k}} \models \varphi_{2j_{1k1}+1,2j_{1k2}+1} \\ q_{a_{11}} \models \varphi_{2j_{211}+1,2j_{212}+1} \\ q_{a_{12}} \models \varphi_{2j_{221}+1,2j_{222}+1} \\ \ldots \\ q_{a_{1k}} \models \varphi_{2j_{2k1}+1,2j_{2k2}+1} \\ \ldots \\ \ldots \\ q_{a_{11}} \models \varphi_{2j_{n11}+1,2j_{n12}+1} \\ q_{a_{12}} \models \varphi_{2j_{n21}+1,2j_{n22}+1} \\ \ldots \\ q_{a_{1k}} \models \varphi_{2j_{nk1}+1,2j_{nk2}+1} \\ q' \models \varphi^0_{2j_1+1,2j_2+2} \end{array}}{q \models \varphi^0_{1,2j_2+2}}$$

$(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}$,

$(q, \sqcap A, q') \in \delta$,

$A = \{a_{11}, a_{12}, \ldots, a_{1k}, a_{21}, a_{22}, \ldots, a_{2l}\}$,

$q_{a_{2i}} = q^{\exists}$ for $1 \le i \le l, j_0 \le j_{pi1} \le j_{pi2} \le j_1$ for all $1 \le p \le n, 1 \le i \le k,$ such that for all $1 \le p \le n$, $\mathbb{N}(\varphi_{2j_{pi1}+1,2j_{pi2}+1}) \cap \mathbb{N}(\varphi_{2j_{pm1}+1,2j_{pm2}+1})$ contains no even natural for all $1 \le i \le k, 1 \le m \le k, i \ne m$, and for all $N \subseteq \mathbb{N}_{2j_0+1,2j_1+1}$ such that $N = \mathbb{N}_{2j_0+1,2j_1+1} \setminus \{2q \mid q \in M\}$ where $M \subseteq \mathbb{N}$, there exists p such that $1 \le p \le n$ and $\bigcup_{1 \le i \le k} \mathbb{N}(\varphi_{2j_{pi1}+1,2j_{pi2}+1}) = N$, and $\neg\phi_{2j_0+1} \implies \phi_{2j_0+2}$, $\neg\phi_{2j_0+3} \implies \phi_{2j_0+4}$, $\ldots$, $\neg\phi_{2j_1-3} \implies \phi_{2j_1-2}$. $\neg\phi_{2j_1-1} \implies \phi_{2j_1}$.

(Fork $\mathcal{U}$ $\exists$)

Figure 3.6. Proof rules for specification checking (part 5)

$$\frac{\begin{array}{l} q_{a_1} \models \varphi'_{2j_{11}+1,2j_{12}+1} \\ q_{a_2} \models \varphi'_{2j_{21}+1,2j_{22}+1} \\ \ldots \\ q_{a_l} \models \varphi_{2j_{l1}+1,2j_{l2}+1} \\ \ldots \\ q_{a_k} \models \varphi'_{2j_{k1}+1,2j_{k2}+1} \\ q' \models \varphi^0_{2j_1+1,2j_2+2} \end{array}}{q \models \varphi^0_{1,2j_2+2}}$$

$(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}$,
$(q, \boxplus A, q') \in \delta, A = \{a_1, a_2, \ldots, a_l, \ldots, a_k\}$,
$j_0 \le j_{i1} \le j_{i2} \le j_1$ for all $1 \le i \le k$, such that
$\mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1}) \cap \mathbb{N}(\varphi'_{2j_{m1}+1,2j_{m2}+1})$ contains no
even natural for all $1 \le i \le k, 1 \le m \le k, i \ne m$,
$i \ne l, m \ne l$, and $\mathbb{N}(\varphi_{2j_{l1}+1,2j_{l2}+1}) \cap \mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1})$
contains no even natural for all $1 \le i \le k, i \ne l$,
and $\bigcup_{1 \le i \le k}^{i \ne l} \mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1}) \cup \mathbb{N}(\varphi_{2j_{l1}+1,2j_{l2}+1}) =$
$\mathbb{N}_{2j_0+1,2j_1+1}$.

(Fork-Choice $\mathcal{U}$)

Figure 3.7. Proof rules for specification checking (part 6)

$$\frac{\begin{array}{l} q_{a_{11}} \models \varphi'_{2j_{11}+1,2j_{12}+1} \\ q_{a_{12}} \models \varphi'_{2j_{21}+1,2j_{22}+1} \\ \ldots \\ q_{a_{1l}} \models \varphi_{2j_{l1}+1,2j_{l2}+1} \\ \ldots \\ q_{a_{1k}} \models \varphi'_{2j_{k1}+1,2j_{k2}+1} \\ q' \models \varphi^0_{2j_1+1,2j_2+2} \end{array}}{q \models \varphi^0_{1,2j_2+2}}$$

$(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}$,
$(q, \boxplus A, q') \in \delta$,
$A = \{a_{11}, a_{12}, \ldots, a_{1l}, \ldots, a_{1k}, a_{21}, a_{22}, \ldots, a_{2q}\}$,
$q_{a_{2i}} = q^{\exists}$ for all $1 \le i \le q, j_0 \le j_{i1} \le j_{i2} \le j_1$
for all $1 \le i \le k,$ such that
$\mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1}) \cap \mathbb{N}(\varphi'_{2j_{m1}+1,2j_{m2}+1})$ contains no
even natural for all $1 \le i \le k, 1 \le m \le k, i \ne m,$
$i \ne l, m \ne l,$ and $\mathbb{N}(\varphi_{2j_{l1}+1,2j_{l2}+1}) \cap \mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1})$
contains no even natural for all $1 \le i \le k, i \ne l,$
and $\bigcup_{1 \le i \le k}^{i \ne l} \mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1}) \cup \mathbb{N}(\varphi_{2j_{l1}+1,2j_{l2}+1}) =$
$\mathbb{N}_{2j_0+1,2j_1+1}$.

(Fork-Choice $\mathcal{U}$ $\exists_1$)

$$\frac{\begin{array}{l} q' \models \varphi^0_{2j_0+1,2j_2+2} \\ q' \models \varphi^0_{2j_0+3,2j_2+2} \\ \ldots \\ q' \models \varphi^0_{2j_2-1,2j_2+2} \\ q' \models \varphi^0_{2j_2+1,2j_2+2} \end{array}}{q \models \varphi^0_{1,2j_2+2}}$$

$(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}$,
$(q, \boxplus A, q') \in \delta, A = \{a_1, a_2, \ldots, a_k\}$,
$q_{a_i} = q^{\exists}$ for all $1 \le i \le k, j_0 \le j_1,$ and
$\neg\phi_{2j_0+1} \implies \phi_{2j_0+2}$,
$\neg\phi_{2j_0+3} \implies \phi_{2j_0+4}$,
$\ldots,$
$\neg\phi_{2j_2-1} \implies \phi_{2j_2}$.
$\neg\phi_{2j_2+1} \implies \phi_{2j_2+2}$.

(Fork-Choice $\mathcal{U}$ $\exists_2$)

Figure 3.8. Proof rules for specification checking (part 7)

$$\frac{}{q \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge \ldots \mathcal{U}\, (\phi_{2j} \wedge \Box\phi_{2j+1}))} \quad \begin{array}{l} \phi_{2i} = \bigwedge_{a \in A_i}(\neg a) \\ \text{where } A_i \subseteq \mathcal{A}, 1 \leq i \leq j, \\ \phi_{2j+1} = \bigwedge_{a \in A_{2j+1}}(\neg a) \\ \text{where } A_{2j+1} \subseteq \mathcal{A}, \\ \text{and } q \text{ is a return location.} \end{array} \qquad \text{(Return } \Box\text{)}$$

$$\frac{q' \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge \ldots \mathcal{U}\, (\phi_{2j} \wedge \Box\phi_{2j+1}))}{q \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge \ldots \mathcal{U}\, (\phi_{2j} \wedge \Box\phi_{2j+1}))} \quad (q, \epsilon, q') \in \delta \cup \delta^{\forall}. \qquad (\epsilon\Box)$$

$$\frac{\begin{array}{l} q_a \models \phi_{2j_0+1} \,\mathcal{U}\, (\phi_{2j_0+2} \wedge \ldots \mathcal{U}\, (\phi_{2j_1} \wedge \Box\phi_{2j_1+1})) \\ q' \models \phi_{2j_1+1} \,\mathcal{U}\, (\phi_{2j_1+2} \wedge \ldots \mathcal{U}\, (\phi_{2j_2} \wedge \Box\phi_{2j_2+1})) \end{array}}{q \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge \ldots \mathcal{U}\, (\phi_{2j_2} \wedge \Box\phi_{2j_2+1}))} \quad \begin{array}{l} (a \ \wedge \neg \bigvee(\mathcal{A} \setminus a)) \Rightarrow \\ (\bigwedge_{0 \leq i \leq j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \\ (q, a, q') \in \delta. \end{array} \qquad \text{(Call } \Box\text{)}$$

$$\frac{\begin{array}{l} q' \models \phi_{2j_0+1} \,\mathcal{U}\, (\phi_{2j_0+2} \wedge \ldots \mathcal{U}\, (\phi_{2j_2} \wedge \Box\phi_{2j_2+1})) \\ q' \models \phi_{2j_0+3} \,\mathcal{U}\, (\phi_{2j_0+4} \wedge \ldots \mathcal{U}\, (\phi_{2j_2} \wedge \Box\phi_{2j_2+1})) \\ \ldots \\ q' \models \phi_{2j_2-1} \,\mathcal{U}\, (\phi_{2j_2} \wedge \Box\phi_{2j_2+1})) \\ q' \models \Box\phi_{2j_2+1})) \end{array}}{q \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge \ldots \mathcal{U}\, (\phi_{2j_2} \wedge \Box\phi_{2j_2+1}))} \quad \begin{array}{l} (a \ \wedge \neg \bigvee(\mathcal{A} \setminus a)) \Rightarrow \\ (\bigwedge_{0 \leq i \leq j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \\ (q, a, q') \in \delta, q_a = q^{\exists}, \\ \neg\phi_{2j_0+1} \implies \phi_{2j_0+2}, \\ \neg\phi_{2j_0+3} \implies \phi_{2j_0+4}, \\ \ldots, \\ \neg\phi_{2j_2-3} \implies \phi_{2j_2-2}. \\ \neg\phi_{2j_2-1} \implies \phi_{2j_2}. \end{array} \qquad \text{(Call } \Box\exists\text{)}$$

Figure 3.9. Proof rules for specification checking (part 8)

$$\frac{\begin{array}{l} q_{a_1} \models \varphi_{2j_{11}+1,2j_{12}+1} \\ q_{a_2} \models \varphi_{2j_{21}+1,2j_{22}+1} \\ \cdots \\ q_{a_k} \models \varphi_{2j_{k1}+1,2j_{k2}+1} \\ q' \models \varphi_{2j_1+1,2j_2+1} \end{array}}{q \models \varphi_{1,2j_2+1}}$$

$$\begin{array}{l} (\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \\ (q, \sqcap A, q') \in \delta, A = \{a_i \mid 1 \le i \le k\}, \\ j_0 \le j_{i1} \le j_{i2} \le j_1 \text{ for all } 1 \le i \le k, \text{ such that} \\ \mathbb{N}(\varphi_{2j_{i1}+1,2j_{i2}+1}) \cap \mathbb{N}(\varphi_{2j_{m1}+1,2j_{m2}+1}) \text{ contains no} \\ \text{even natural for all } 1 \le i \le k, 1 \le m \le k, i \ne m, \\ \text{and } \bigcup_{1 \le i \le k} \mathbb{N}(\varphi_{2j_{i1}+1,2j_{i2}+1}) = \mathbb{N}_{2j_0+1,2j_1+1}. \end{array}$$

(Fork $\sqcap$)

Figure 3.10. Proof rules for specification checking (part 9)

dom only to the extent allowed by nondeterminism in the interface, which it uses to attempt to violate the specification. The unsupported actions in the interface namespace, corresponding to actions that will be supported in refinement, play the game attempting to make the interface violate the specification. If the environment has a strategy to win the game, we say the interface satisfies the specification. This means that, irrespective of behavioral nondeterminism, and how the unimplemented actions in the interface namespace are later implemented, the interface can always be used in a way that will ensure the desired property is satisfied. For instance, if vendor A is using a service S with an interface I provided by vendor B, vendor A will receive a guarantee that as long as it respects a protocol (given by the environment's winning strategy for the property desired by A on interface I), no matter what internal refinements are made to the implementation of S by vendor B, the property desired by A will continue to hold true.

The proof rules in Figure 3.2 encode the definitions for boolean operators, and

$$\frac{\begin{array}{l} q_{a_{11}} \models \varphi_{2j_{111}+1,2j_{112}+1} \\ q_{a_{12}} \models \varphi_{2j_{121}+1,2j_{122}+1} \\ \ldots \\ q_{a_{1k}} \models \varphi_{2j_{1k1}+1,2j_{1k2}+1} \\ q_{a_{11}} \models \varphi_{2j_{211}+1,2j_{212}+1} \\ q_{a_{12}} \models \varphi_{2j_{221}+1,2j_{222}+1} \\ \ldots \\ q_{a_{1k}} \models \varphi_{2j_{2k1}+1,2j_{2k2}+1} \\ \ldots \\ \ldots \\ q_{a_{11}} \models \varphi_{2j_{n11}+1,2j_{n12}+1} \\ q_{a_{12}} \models \varphi_{2j_{n21}+1,2j_{n22}+1} \\ \ldots \\ q_{a_{1k}} \models \varphi_{2j_{nk1}+1,2j_{nk2}+1} \\ q' \models \varphi_{2j_1+1,2j_2+1} \end{array}}{q \models \varphi_{1,2j_2+1}}$$

$(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}$,

$(q, \sqcap A, q') \in \delta, A = \{a_{11}, a_{12}, \ldots, a_{1k}, a_{21}, a_{22}, \ldots, a_{2l}\}$,

$q_{a_{2i}} = q^{\exists}$ for $1 \le i \le l, j_0 \le j_{pi1} \le j_{pi2} \le j_1$ for all $1 \le p \le n, 1 \le i \le k,$ such that for all $1 \le p \le n$, $\mathbb{N}(\varphi_{2j_{pi1}+1,2j_{pi2}+1}) \cap \mathbb{N}(\varphi_{2j_{pm1}+1,2j_{pm2}+1})$ contains no even natural for all $1 \le i \le k, 1 \le m \le k, i \ne m$, and for all $N \subseteq \mathbb{N}_{2j_0+1,2j_1+1}$ such that $N = \mathbb{N}_{2j_0+1,2j_1+1} \setminus \{2q \mid q \in M\}$ where $M \subseteq \mathbb{N}$, there exists p such that $1 \le p \le n$ and $\bigcup_{1 \le i \le k} \mathbb{N}(\varphi_{2j_{pi1}+1,2j_{pi2}+1}) = N,$ and

$\neg\phi_{2j_0+1} \implies \phi_{2j_0+2}$,

$\neg\phi_{2j_0+3} \implies \phi_{2j_0+4}$,

$\ldots$,

$\neg\phi_{2j_1-3} \implies \phi_{2j_1-2},$ and

$\neg\phi_{2j_1-1} \implies \phi_{2j_1}$.

(Fork $\Box\exists$)

Figure 3.11. Proof rules for specification checking (part 10)

$$\frac{\begin{array}{l} q_{a_1} \models \varphi'_{2j_{11}+1,2j_{12}+1} \\ q_{a_2} \models \varphi'_{2j_{21}+1,2j_{22}+1} \\ \ldots \\ q_{a_l} \models \varphi_{2j_{l1}+1,2j_{l2}+1} \\ \ldots \\ q_{a_k} \models \varphi'_{2j_{k1}+1,2j_{k2}+1} \\ q' \models \varphi_{2j_1+1,2j_2+1} \end{array}}{q \models \varphi_{1,2j_2+1}}$$

$(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \leq i \leq j_0} \phi_{2i}) \wedge \phi_{2j_0+1}$,
$(q, \boxplus A, q') \in \delta, A = \{a_1, a_2, \ldots, a_l, \ldots, a_k\}$,
$0 \leq j_0 \leq j_{i1} \leq j_{i2} \leq j_1 \leq j_2$for all $1 \leq i \leq k$,
such that $\mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1}) \cap \mathbb{N}(\varphi'_{2j_{m1}+1,2j_{m2}+1})$
contains no even natural for all $1 \leq i \leq k$,
$1 \leq m \leq k, i \neq m, i \neq l, m \neq l,$ and
$\mathbb{N}(\varphi_{2j_{l1}+1,2j_{l2}+1}) \cap \mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1})$contains no
even natural for all $1 \leq i \leq k, i \neq l,$ and
$\bigcup_{1 \leq i \leq k}^{i \neq l} \mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1}) \cup \mathbb{N}(\varphi_{2j_{l1}+1,2j_{l2}+1}) =$
$\mathbb{N}_{2j_0+1,2j_1+1}$.

(Fork-Choice $\Box$)

Figure 3.12. Proof rules for specification checking (part 11)

$$\frac{\begin{array}{l} q_{a_{11}} \models \varphi'_{2j_{11}+1,2j_{12}+1} \\ q_{a_{12}} \models \varphi'_{2j_{21}+1,2j_{22}+1} \\ \cdots \\ q_{a_{1l}} \models \varphi_{2j_{l1}+1,2j_{l2}+1} \\ \cdots \\ q_{a_{1k}} \models \varphi'_{2j_{k1}+1,2j_{k2}+1} \\ q' \models \varphi_{2j_1+1,2j_2+1} \end{array}}{q \models \varphi_{1,2j_2+1}}$$

$(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \leq i \leq j_0} \phi_{2i}) \wedge \phi_{2j_0+1}$,
$(q, \boxplus A, q') \in \delta$,
$A = \{a_{11}, a_{12}, \ldots, a_{1l}, \ldots, a_{1k}, a_{21}, a_{22}, \ldots, a_{2q}\}$,
$q_{a_{2i}} = q^{\exists}$ for all $1 \leq i \leq q$,
$0 \leq j_0 \leq j_{i1} \leq j_{i2} \leq j_1 \leq j_2$ for all $1 \leq i \leq k$,
such that $\mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1}) \cap \mathbb{N}(\varphi'_{2j_{m1}+1,2j_{m2}+1})$
contains no even natural for all $1 \leq i \leq k$,
$1 \leq m \leq k, i \neq m, i \neq l, m \neq l,$ and
$\mathbb{N}(\varphi_{2j_{l1}+1,2j_{l2}+1}) \cap \mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1})$contains no
even natural for all $1 \leq i \leq k, i \neq l,$ and
$\bigcup_{1 \leq i \leq k}^{i \neq l} \mathbb{N}(\varphi'_{2j_{i1}+1,2j_{i2}+1}) \cup \mathbb{N}(\varphi_{2j_{l1}+1,2j_{l2}+1}) =$
$\mathbb{N}_{2j_0+1,2j_1+1}$.

(Fork-Choice $\Box\exists_1$)

Figure 3.13. Proof rules for specification checking (part 12)

$$\frac{\begin{array}{l} q' \models \varphi_{2j_0+1,2j_2+1} \\ q' \models \varphi_{2j_0+3,2j_2+1} \\ q' \models \varphi_{2j_0+5,2j_2+1} \\ \ldots \\ q' \models \varphi_{2j_2-1,2j_2+1} \\ q' \models \Box\phi_{2j_2+1} \end{array}}{q \models \varphi_{1,2j_2+1}}$$

$(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}$,

$(q, \boxplus A, q') \in \delta, A = \{a_1, a_2, \ldots, a_k\}, q_{a_i} = q^{\exists}$ for all $1 \le i \le k$.

$0 \le j_0 \le j_2$for all $1 \le i \le k$,

and

$\neg\phi_{2j_0+1} \implies \phi_{2j_0+2}$,

$\neg\phi_{2j_0+3} \implies \phi_{2j_0+4}$,

$\ldots$,

$\neg\phi_{2j_2-3} \implies \phi_{2j_2-2}$. and

$\neg\phi_{2j_2-1} \implies \phi_{2j_2}$.

(Fork-Choice $\Box\exists_2$)

Figure 3.14. Proof rules for specification checking (part 13)

the silentness of ϵ-transitions. The remaining proof rules encode the strategies of the two players to try to satisfy the temporal logic formulas used in specifications, and the additional (helper) formulas introduced in our specification-checking scheme. The semantics of the various sequential and parallel composition operators allowed in protocol interfaces are captured in separate proof rules that apply when those operators are concerned.

The proof rule labeled "Reached $\mathcal{U}^0$" in Figure 3.3 states that if the until-formula is already satisfied in the process of an action or a set of actions being invoked, then no further premises need to be satisfied in order to reach the conclusion. The next rule ("Reached $\mathcal{U}_1^+$") states that if a prefix of the until-formula is satisfied in the process of the invocation of an action c, and the location q_c from which execution continues is such that q_c satisfies the rest of the until formula starting from that point, then the until-formula is satisfied by the location q that invoked the action c. The next rule ("Reached $\mathcal{U}_2^+$") handles the analogous case where a set of actions are invoked in parallel using the $\sqcap$ or $\boxplus$ operators, and a prefix of the desired until-formula is satisfied in the process of that invocation; in this case, we require the set of actions invoked in parallel to be such that there is an execution-interleaving of their resultant threads that satisfies the rest of the until-formula. Note that an action a_l is chosen to complete the process of satisfying the desired until-formula; the remaining actions are allowed to only satisfy a formula of the form $x\,\mathcal{U}\,\mathrm{T}$, i.e. they may exhibit any arbitrary behavior after they have shown the desired behavior up to a certain point (depending on the formula x). Note also that these proof rules do not apply to the location $q^{\exists}$; the environment will never satisfy these temporal logic formulas as far as these proof rules are concerned because there is nothing in the proof rules or side-conditions to force it to do so.

The next two proof rules ("Call $\mathcal{U}$" and "Call $\mathcal{U}$ $\exists$") in Figure 3.4 involve the case when a single action a is invoked satisfying a prefix of the desired until-formula, and then the behavior resulting from the action invocation satisfies only a part of the rest of the until-formula, requiring the location q' (from which execution continues after control returns from the execution of a) to satisfy the final part. In the first of these two rules, which applies if the action a is not an action outside the interface namespace and thus supported by the environment, the invocation of a satisfies a prefix of the desired until-formula, the location q_a satisfies the first part of the remainder, and the location q' satisfies the final part, as expected. The second rule applies when the action a is supported by the environment. Then, after only a prefix of the desired until-formula is satisfied in the process of invoking action a, the only way the environment can be prevented from violating the rest of the until-formula (let's denote it x) is by the series of implications in the side-condition that make it impossible for the until-formula to be violated by any finite trace. However, the environment cannot run for ever as we require every action invocation to eventually terminate execution. The environment however, can choose to satisfy any arbitrary prefix of the remainder x of the until-formula, and the location q' therefore should be able to carry on starting from an arbitrary point in the remainder x.

The next proof rule ("Fork $\mathcal{U}$") in Figure 3.5 handles the case where a set of actions are invoked in parallel (using the $\sqcap$ operator) such that the invocation satisfies a prefix of the desired until-formula, the execution of the parallel invocations can be interleaved to satisfy a part of the remainder, and the location q' (where control arrives after the parallel execution is completed) can carry on to satisfy the rest of the remainder. This proof rule can be applied when none of the actions invoked in parallel are outside the interface namespace.

The next proof rule ("Fork $\mathcal{U}$ $\exists$") in Figure 3.6 is a variation of the above rule that applies when some (or all) of the actions are supported by the environment. Then, since control is given to the environment at some point before the until-formula is satisfied in its entirety, once again we need the series of implications in the side-condition to make sure no finite trace the environment can generate can violate any part of the remainder of the until-fomula that remains to be satisfied when the environment can gain control. Then, we require that the non-environment actions (if any) that are invoked should be able to be interleaved among themselves to satisfy any arbitrary part of the remainder that need to be satisfied (because the environment may or may not choose to satisfy some parts of that remainder when it gains control). Finally, the final part of the remainder needs to be satisfied by the location q'. Note that the enviroment actions must be given control (by the scheduler) at some point in order to allow the execution of the parallel invocation to complete so that control may pass to q'.

The next proof rule ("Fork-Choice $\mathcal{U}$") in Figure 3.7 applies to the case where a number of non-environment actions are invoked in parallel (using the $\boxplus$ operator) such that the invocation satisfies a prefix of the desired until-formula, some prefixes of the traces generated by the execution of the actions can be interleaved by the scheduler to satisfy a part of the remainder such that at least one of the traces (the one corresponding to action a_l) is used till completion (and the rest of the traces are aborted at that point; a freedom available to the scheduler in case of parallel executions done using the $\boxplus$ operator, but not in case of parallel executions done using the $\sqcap$ operator); and the location q' satisfies the rest of the remainder.

The next two proof rules ("Fork-Choice $\mathcal{U}$ $\exists_1$" and "Fork-Choice $\mathcal{U}$ $\exists_2$") in Figure 3.8 are variations of the above rule to handle the cases where a number of actions

are invoked in parallel (using the $\boxplus$ operator), and respectively at least some, or all, of those actions are outside the interface namespace, supported by the environment. In the former case, (apart from their use in instantaneously satisfying a prefix of the desired until-formula) the environment actions are completely ignored; the scheduler never needs to schedule any of them to execute at all, because the execution of these environment actions would not help at all and would seek only to try to violate the temporal logic formula that the interface seeks to satisfy. The scheduler is able to avoid scheduling these environment actions because there are other actions in the interface namespace one of which (the one corresponding to action a_l) runs to completion, allowing the execution of the rest (including the environment actions which were never scheduled to execute at all) to be aborted. As in the previous rule, if the behavior of those actions satisfies a prefix of the desired remainder of the until-formula, and the location q' is able to satisfy the final part, the desired conclusion follows.

However, there may be cases in which the set of actions invoked in parallel (using the $\boxplus$ operator) are *all* environment actions. In that case the second rule ("Fork-Choice $\mathcal{U}$ $\exists_2$") of Figure 3.8 applies. We need to let at least one of the environment actions to completion. Thus, we need the series of implications in the side-condition that ensures that the environment cannot generate any finite trace that violates the desired remainder of the until-formula. Finally, q' is ready to satisfy the desired remainder starting from any arbitrary point in the specification the environment chooses to return control.

The rest of the proof rules encode the strategies of the two players to try to satisfy or violate respectively, the helper temporal logic formulas introduced in order

to define the previous proof rules. Exactly the same considerations apply in these rules.

3.5 Case Study

We present the following case study on using our formalism to formally model a web-based sales system $\mathcal{P}$ that is built using the Amazon.com E-Commerce Services (ECS) platform.

The system has the set of web methods $\mathcal{M}_{\texttt{Local}} = \{\texttt{BeginTransaction}, \texttt{ContinueTransaction}, \texttt{FindItems}, \texttt{BrowseNewItems}, \texttt{ProcessPayment}, \texttt{ShipItems}\}$. It uses the web methods provided by the Amazon.com ECS platform represented by the set $\mathcal{M}_{\texttt{Amazon}} = \{\texttt{ItemSearch}, \texttt{CartCreate}, \texttt{CartAdd}, \texttt{CartModify}, \texttt{CheckOut}\}$. The set of instances $\mathcal{I} = \{s\} \cup \texttt{ASIN} \cup \texttt{CARTID} \cup \texttt{CATID}$ is used to characterize web method invocations, where s denotes successful completion of a web method invocation, `ASIN` is the set of Amazon Standard Identification Numbers used to represent items for sale, `CARTID` is the set of Cart Identifiers used by the Amazon.com ECS platform to distinguish between the virtual shopping carts assigned to various online shopping customers, and `CATID` is the set of Category Identifiers used by the ECS platform to represent various categories of items, such as Books, Music, Movies, Garments, etc. The set of actions $\mathcal{A}$ is given by $\mathcal{A} = (\mathcal{M}_{\texttt{Local}} \cup \mathcal{M}_{\texttt{Amazon}} \cup F) \times \mathcal{I}$, where F is a set of fresh symbols.

The web-based sales system $\mathcal{P}$ is now defined formally as a protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$, where the partial function $\mathcal{R} : \mathcal{A} \rightarrow Q$ maps a supported action to a location, which we denote below by writing an action in front of the location that the action is mapped to, and the protocol automaton $\mathcal{H} = (Q, \delta)$ is

represented by giving the transition relation δ of the automaton as a sequence of triples (q, term, q'). Note that q_0 is a return location.

$\mathcal{P}$:

$$
\begin{array}{ll}
\langle \texttt{BeginTransaction}, s\rangle \mapsto & (q_1, \langle \texttt{CartCreate}, c\rangle, q_2) \\
 & (q_2, \langle \texttt{ContinueTransaction}, c\rangle, q_0) \\
\langle \texttt{ContinueTransaction}, c\rangle \mapsto & (q_3, \boxplus\{\langle a, s\rangle, \langle b, s\rangle, \langle c, s\rangle, \langle d, s\rangle\}, q_0) \\
\langle a, s\rangle \mapsto & (q_4, \langle \texttt{BrowseNewItems}, c\rangle, q_0) \\
\langle b, s\rangle \mapsto & (q_5, \langle \texttt{CartModify}, c\rangle, q_0) \\
\langle c, s\rangle \mapsto & (q_6, \langle \texttt{CheckOut}, c\rangle, q_0) \\
\langle d, s\rangle \mapsto & (q_7, \langle \texttt{ContinueTransaction}, c\rangle, q_0) \\
\langle \texttt{BrowseNewItems}, c\rangle \mapsto & (q_8, \langle \texttt{FindItems}, s\rangle, q_0) \\
\langle \texttt{BrowseNewItems}, c\rangle \mapsto & (q_9, \langle \texttt{FindItems}, s\rangle, q_{10}) \\
 & (q_{10}, \langle \texttt{CartAdd}, c\rangle, q_0) \\
\langle \texttt{FindItems}, s\rangle \mapsto & (q_{11}, \boxplus\{\langle \texttt{ItemSearch}, c1\rangle, \langle \texttt{ItemSearch}, c2\rangle\}, q_0)
\end{array}
$$

The execution begins with the invocation of the web method `BeginTransaction`, which is implemented by the service using the web method `CartCreate` provided by the Amazon ECS, which creates a new shopping cart and returns the cart identifier c to the service $\mathcal{P}$. The service $\mathcal{P}$ then allows the user (the customer, i.e., the person using the service $\mathcal{P}$ to buy items online) to search for new items and optionally add them to the shopping cart, modify the contents of the shopping cart, and check out. The former three actions can be done in parallel: the user can open multiple browser windows, through which multiple search, cart-add, and cart-modify transactions can be run in parallel, all of which would be dealing with the multi-threaded shopping cart object provided by the ECS platform. When the web method `CheckOut` is invoked, Amazon.com atomically inspects the contents of the user's cart, charges the

customer for the total price, and ships the items, completing the sale. Thus, note that in particular, the user pays for exactly those items that get shipped.

Now, let us consider a modification of the system above to account for a change in the business model of our web-based shop. Previously, the shop depended upon Amazon.com to take care of billing the customer and shipping products. Now, to establish a closer relationship with the customer, our web-based shop decides to implement billing and shipping themselves in their own local check-out method `LocalCheckOut` and helpers `ComputePrice` and `ShipItems`, and continue using the Amazon.com ECS platform for only its product search services and shopping cart implementation.

The modified system is $\mathcal{P}' = (\mathcal{N}, \mathcal{J}', \mathcal{K}', \mathcal{H}', \mathcal{R}')$ with the following additional supported actions $\langle \texttt{LocalCheckOut}, c\rangle$, $\langle \texttt{ComputePrice}, \texttt{OK}\rangle$, and $\langle \texttt{ShipItems}, \texttt{OK}\rangle$, and the modified partial function $\mathcal{R}'$ is obtained from $\mathcal{R}$ by adding the mappings indicated below, and the modified protocol automaton $\mathcal{H}'$ obtained from $\mathcal{H}$ by replacing the invocation of $\langle \texttt{CheckOut}, c\rangle$ in $\mathcal{H}$ with that of $\langle \texttt{LocalCheckOut}, c\rangle$, and adding to $\mathcal{H}$ the locations and transitions represented as follows:

$$\begin{aligned}
\langle \texttt{LocalCheckOut}, c\rangle \mapsto \quad & (q_{12}, \langle \texttt{ComputePrice}, \texttt{OK}\rangle, q_{13}) \\
& (q_{13}, \langle \texttt{ShipItems}, \texttt{OK}\rangle, q_0) \\
\langle \texttt{ComputePrice}, \texttt{OK}\rangle \mapsto \quad & (q_{14}, \epsilon, q_0) \\
\langle \texttt{ShipItems}, \texttt{OK}\rangle \mapsto \quad & (q_{15}, \epsilon, q_0)
\end{aligned}$$

This simple modification has indeed introduced a severe error in our application. To appreciate this, let us consider the specification $\psi = \langle \texttt{BeginTransaction}, s\rangle \not\rightsquigarrow (\mathrm{T}\ \ \mathcal{U}\ \ (\langle \texttt{ComputePrice}, \texttt{OK}\rangle \wedge (\mathrm{T}\ \ \mathcal{U}\ \ (\langle \texttt{CartAdd}, c\rangle \vee \langle \texttt{CartModify}, c\rangle) \wedge (\mathrm{T}\ \ \mathcal{U}\ \ \langle \texttt{ShipItems}, \texttt{OK}\rangle))))$. Intuitively, this specification represents the question "Is an execution starting with the invocation of $\langle \texttt{BeginTransaction}, s\rangle$ possible on which a

cart-add or cart-modify transaction is successfully completed after price computation has been successfully invoked on the cart but before the items in the cart have been shipped?" Note that $\mathcal{P}' \not\models \psi$. Thus, it is possible for an execution trace to occur (involving a race condition), in which the customer is able to make one or more cart-add or cart-modify transaction(s) after the price has been computed, but before the items in the shopping cart have been shipped. On such an execution trace, the customer could pay for items that never get shipped, or vice versa; both cases involving incorrect behavior.

We note that the problem can be resolved by getting rid of some of the parallelism afforded by the original design. We modify the service above accordingly, and obtain the following web-service interface $\mathcal{P}'' = (\mathcal{N}, \mathcal{J}', \mathcal{K}', \mathcal{H}'', \mathcal{R}'')$, where the modified partial function $\mathcal{R}''$ is obtained from $\mathcal{R}'$ by adding the mappings indicated below, and the modified protocol automaton $\mathcal{H}''$ obtained from $\mathcal{H}'$ by replacing the locations and transitions for $\langle a, s\rangle$, $\langle b, s\rangle$, $\langle c, s\rangle$, and $\langle d, s\rangle$ in $\mathcal{H}'$ with the following:

$$
\begin{aligned}
\langle a, s\rangle \mapsto\ & (q_{16}, \langle \texttt{BrowseNewItems}, c\rangle, q_{17}) \\
& (q_{17}, \langle \texttt{ContinueTransaction}, c\rangle, q_0) \\
\langle a, s\rangle \mapsto\ & (q_{18}, \langle \texttt{CartModify}, c\rangle, q_{17}) \\
\langle a, s\rangle \mapsto\ & (q_{19}, \langle \texttt{LocalCheckOut}, c\rangle, q_0) \\
\langle b, s\rangle \mapsto\ & (q_{20}, \epsilon, q_0) \\
\langle c, s\rangle \mapsto\ & (q_{21}, \epsilon, q_0) \\
\langle d, s\rangle \mapsto\ & (q_{22}, \epsilon, q_0)
\end{aligned}
$$

We note that $\mathcal{P}'' \models \psi$. However, how would we know that the modified service $\mathcal{P}''$ can be safely substituted in place of the service $\mathcal{P}'$? To this end we can simply check refinement by asking the question if $\mathcal{P}'' \preccurlyeq \mathcal{P}'$. The answer turns out to be Yes,

which means that the protocol interface $\mathcal{P}''$ may be safely substituted in place of $\mathcal{P}'$ in any arbitrary context. In other words, we can infer that in the process of obtaining $\mathcal{P}''$ by fixing the bug in protocol interface $\mathcal{P}'$ we have not introduced any additional (un-intended) behavior into $\mathcal{P}''$ that could lead to new errors (violations of any safety properties we may be interested in), as yet undiscovered. Thus, we conclude that the final service is the desired result.

Acknowledgements

The work reported in this chapter was conducted jointly with Prof Dirk Beyer, Prof Thomas A. Henzinger, and Prof Sanjit A. Seshia. This research was funded by Prof Henzinger supported in part by the ONR grant N00014-02-1-0671, the NSF ITR CHESS grant CCR-0225610, the NSF grant CCR-0234690, and by the Swiss National Science Foundation. This chapter is based on a paper [25] presented at WWW 2005, published by ACM[2], copyright held by the International World Wide Web Conference Committee (IW3C2), 2005; the EPFL Technical Report Number MTC-REPORT-2007-002 [26][3] co-authored with Prof Dirk Beyer and Prof Tom Henzinger; and a paper [27] presented at ICWS 2007, copyright held by IEEE[4], 2007.

[2] `http://doi.acm.org/10.1145/1060745.1060770`

[3] A preliminary version of this Technical Report was presented at the First International Workshop on Foundations of Interface Technologies (FIT), held on August 21, 2005, in San Francisco, CA.

[4] `http://doi.ieeecomputersociety.org/10.1109/ICWS.2007.32`

Algorithm 2 CheckSpec$(\mathcal{P}, a, B, C)$

Input: Protocol interface $\mathcal{P} = (\mathcal{N}, \mathcal{J}, \mathcal{K}, \mathcal{H}, \mathcal{R})$,

Action $a \in \mathcal{A}$, and formula φ over actions in $\mathcal{A}$

Output: YES if $\mathcal{P}$ satisfies $a \not\rightsquigarrow \varphi$, NO otherwise

Variables: Set of judgements S, boolean `done`

1: `done` := F

2: **while** (¬ `done`) **do**

3: `done` := T

4: **for each** location q of automaton $\mathcal{H}$ **do**

5: // Try to prove $q \models \phi$.

6: **if** all premises of a rule for a consequent c of the form $q \models \phi$ are in S **then**

7: $S := S \cup c$

8: `done` := F

9: // Try to prove $q \models \varphi_{i,j}$.

10: **if** all premises of a rule for a consequent of the form $q \models \varphi_{i,j}$ are in S **then**

11: $S := S \cup c$

12: `done` := F

13: // Try to prove $q \models \varphi^0_{i,j}$.

14: **if** all premises of a rule for a consequent of the form $q \models \varphi^0_{i,j}$ are in S **then**

15: $S := S \cup c$

16: `done` := F

end

17: **if** $q_a \models \varphi \quad \in \ S$ **then**

18: **return** NO

end

19: **return** YES

Chapter 4

Resource Interfaces

4.1 Introduction

In component-based design, a central notion is that of *interfaces*: an interface should capture those facts about a component that are necessary and sufficient for determining if a collection of components fits together. The formal notion of interface, then, depends on what "fitting together" means. In a simple case, an interface exposes only type information about the component's inputs and outputs, and "fitting together" is determined by type checking. In a more ambitious case, an interface may expose also temporal information about inputs and outputs. For example, the temporal interface of a file server may specify that the `open_file` method must be called before the `read_file` method is invoked. If a client, instead, calls `read_file` before `open_file`, then an interface violation occurs. In [54], we argued that temporal interfaces are *games*. There are two players, Input and Output, and an objective, namely, the absence of interface violations. Then, an interface is *well-formed* if the corresponding component can be used in some environment; that is, player Input has

a strategy to achieve the objective. Moreover, two interfaces are *compatible* if the corresponding components can be used together in some environment; that is, the composition of the two games is well-formed, and specifies the composite interface.

Here, we develop the theory of interfaces as games further, by introducing interfaces that expose resource information. Consider, for example, components whose power consumption varies. We model the interface of such a component as a control flow graph whose states are labeled with integers, which represent the power consumption while control is at that state. For instance, in the thread-based programming model for robot motor control presented in Section 5, the power consumption of a program region depends on how many motors and other devices are active. Now suppose that we want to put together two threads, each of which consumes power, but the overall amount of available peak power is limited to a fixed amount Δ. The threads are controlled by a scheduler, which at all times determines the thread that may progress. Then the two threads are Δ-*compatible* if the scheduler has a strategy to let them progress in a way so that their combined power consumption never exceeds Δ. In more detail, the game is set up as follows: player Input is the scheduler, and player Output is the composition of the two threads. At each round of the game, player Input determines which thread may proceed, and player Output determines the successor state in the control flow graph of the scheduled thread. In this game, in order to avoid a safety violation (power consumption greater than Δ), player Input may not let any thread progress. To rule out such trivial schedules, one may augment the safety objective with a secondary, liveness objective, say, in the form of a Büchi condition, which specifies that the scheduler must allow each thread to progress infinitely often. The resulting compatibility check, then, requires the solution of a Büchi

game: the two threads are Δ-compatible iff player Input has a strategy to satisfy the Büchi condition without exceeding the power threshold Δ.

The basic idea of formalizing interfaces as such *Büchi node limit games* on integer-labeled graphs has many applications besides power consumption. For example, access to a mutex resource can be modeled by state labels 0 and 1, where 1 represents usage of the resource. Then, if we choose $\Delta = 1$, two or more threads are Δ-compatible if at any time at most one of the threads uses the resource. In Section 5, we will also present an interface model for the clients of a wireless network, where each state label represents the number of active messages at a node of the network, and Δ represents the buffer size. In this example, the Δ-compatibility check synthesizes not a scheduling strategy but a routing protocol that keeps the message buffers from overflowing.

A wide variety of other formalisms for the modeling and analysis of resource constraints have been proposed in the literature (e.g., [89, 107, 124, 135]). The essential difference between these papers and our work is that we pursue a compositional approach, in which the models and analysis techniques are based on games. Once resource interfaces are modeled as games on graphs with integer labels, in addition to the *boolean* question of Δ-compatibility, for fixed Δ, we can also ask a corresponding *quantitative* question about resource requirements: What is the minimal resource level (peak power, buffer size, etc.) Δ necessary for two or more given interfaces to be compatible? To formalize the quantitative question, we need to define the *value* of an outcome of the game, which is the infinite sequence of states that results from playing the game for an infinite number of rounds. For Büchi node limit games, the value of an outcome is the supremum of the power consumption over all states of the outcome. The player Input (the scheduler) tries to minimize the value, while the

player Output (the thread set) tries to maximize. The quantitative question, then, asks for the inf-sup of the value over all player Input and Output strategies.

The node limit interfaces, where an interface violation occurs if a power threshold is exceeded at any one time, provide but one example of how the compatibility of resource interfaces may be defined. We also present a second use of resource interfaces, where a violation occurs when an initially available amount Δ of energy (given, say, by the capacity of a battery) is exhausted. In this case, the value u of a finite outcome is defined as the *sum* (rather than maximum) over all labels of the states of the outcome, and player Input (the scheduler) wins if it can keep $\Delta - u$ nonnegative forever, or until a certain task is achieved. Note that in this game, negative state labels can be used to model a recharging of the energy source. Achieving a task might be modeled again by a Büchi objective, but for variety's sake, we use a quantitative *reward* objective in our formalization of such *path limit interfaces*. For this purpose, we label each state with a second number, which represents a reward, and the objective of player Input is to obtain a total accumulated reward of Λ. The boolean Δ-compatibility question, then, asks if Λ can be obtained from the composition of two interfaces without exceeding the initial energy supply Δ. The corresponding quantitative resource-requirement question asks for the minimum initial energy supply Δ necessary to achieve the fixed reward Λ. Dually, a similar algorithm can be used to determine the maximal achievable reward Λ given a fixed initial energy supply Δ. In particular, if every state offers reward 1, this asks for the maximum runtime of a system (in number of state transitions) that a scheduler can achieve with a given battery capacity.

In the following section we reviews the definitions needed for modeling temporal (resourceless) interfaces as games and then in the following section we add resources

to these games: we introduce integer labels on states to model resource usage, and we define boolean as well as quantitative objective functions on the outcomes of a game. As examples, we define four specific resource-interface theories: node limit games without and with Büchi objectives, and path limit games without and with reward objectives. For these four theories, in the next section we present algorithms for solving the boolean Δ-compatibility and the quantitative resource-requirement questions. These interface theories are also used in the two case studies of the next section, one on scheduling embedded threads for robot control, and the other on routing messages across wireless networks.

4.2 Preliminaries

An *interface* is a system model that represents both the behavior of a component, and the behavior the component expects from its environment [54]. An interface communicates with its environment through input and output variables. The interface consists of a set of states. Associated with each state is an input assumption, which specifies the input values that the component is ready to accept from the environment, and an output guarantee, which specifies the output values that the component can generate. Once the input values are received and the output values are generated, these values cause a transition to a new state. In this way, both input assumptions and output guarantees can change dynamically. Formally, an *assume-guarantee* (A/G) *interface* [55] is a tuple $M = \langle V^i, V^o, Q, \hat{q}, \phi^i, \phi^o, \rho \rangle$ consisting of:

- Two finite sets V^i and V^o of boolean *input* and *output variables*. We require that $V^i \cap V^o = \emptyset$.
- A finite set Q of *states*, including an initial state $\hat{q} \in Q$.

- Two functions ϕ^i and ϕ^o which assign to each state $q \in Q$ a satisfiable predicate $\phi^i(q)$ on V^i, called *input assumption*, and a satisfiable predicate $\phi^o(q)$ on V^o, called *output guarantee*.

- A function ρ which assigns to each pair $q, q' \in Q$ of states a predicate $\rho(q, q')$ on $V^i \cup V^o$, called the *transition guard*. We require that for every state $q \in Q$, we have (1) $(\phi^i(q) \wedge \phi^o(q)) \Rightarrow \bigvee_{q' \in Q} \rho(q, q')$ and (2) $\bigwedge_{q', q'' \in Q}((\rho(q, q') \wedge \rho(q, q'')) \Rightarrow (q' = q''))$. Condition (1) ensures nonblocking; condition (2) ensures determinism.

We refer to the states of M as Q_M, etc. Given a set V of boolean variables, a *valuation* v for V is a function that maps each variable $x \in V$ to a boolean value $v(x)$. A valuation for V^i (resp. V^o) is called an *input* (resp. *output*) *valuation*. We write $\mathbb{V}^i$ and $\mathbb{V}^o$ for the sets of input and output valuations.

Interfaces as games. An interface is best viewed as a game between two players, Input and Output. The game $G_M = \langle Q, \hat{q}, \gamma^i, \gamma^o, \delta \rangle$ associated with the interface M is played on the set Q of states of the interface. At each state $q \in Q$, player Input chooses an input valuation v^i that satisfies the input assumption, and simultaneously and independently, player Output chooses an output valuation v^o that satisfies the output guarantee; that is, at state q the moves available to player Input are $\gamma^i(q) = \{v \in \mathbb{V}^i \mid v \models \phi^i(q)\}$, and the moves available to player Output are $\gamma^o(q) = \{v \in \mathbb{V}^o \mid v \models \phi^o(q)\}$. Then the game proceeds to the state $q' = \delta(q, v^i, v^o)$, which is the unique state in Q such that $(v^i \uplus v^o) \models \rho(q, q')$. The result of the game is a run. A *run* of M is an infinite sequence $\pi = q_0, (v_0^i, v_0^o), q_1, (v_1^i, v_1^o), q_2, \ldots$ of alternating states $q_k \in Q$, input valuations $v_k^i \in \mathbb{V}^i$, and output valuations $v_k^o \in \mathbb{V}^o$, such that for all $k \geq 0$, we have (1) $v_k^i \in \gamma^i(q_k)$ and $v_k^o \in \gamma^o(q_k)$, and (2) $q_{k+1} = \delta(q_k, v_k^i, v_k^o)$. The run

π is *initialized* if $q_0 = \hat{q}$. A *run prefix* is a finite prefix of a run which ends in a state. Given a run prefix π, we write $last(\pi)$ for the last state of π.

In a game, the players choose moves according to strategies. An *input strategy* is a function that assigns to every run prefix π an input valuation in $\gamma^i(last(\pi))$, and an *output strategy* is a function that assigns to every run prefix π an output valuation in $\gamma^o(last(\pi))$. We write Σ^i and Σ^o for the sets of input and output strategies. Given a state $q \in Q$, and a pair $\sigma^i \in \Sigma^i$ and $\sigma^o \in \Sigma^o$ of strategies, the *outcome* of the game *from* q is the run $out(q, \sigma^i, \sigma^o) = q_0, (v_0^i, v_0^o), q_1, (v_1^i, v_1^o), \ldots$ such that (1) $q_0 = q$ and (2) for all $k \geq 0$, we have $v_k^i = \sigma^i(q_0, \ldots, q_k)$ and $v_k^o = \sigma^o(q_0, (v_0^i, v_0^o), q_1, \ldots, q_k)$.

The *size* of the A/G interface M is taken to be the size of the associated game G_M: define $|M| = \sum_{q \in Q} |\gamma^i(q)| \cdot |\gamma^o(q)|$. Since the interface M is specified by predicates on boolean variables, the size $|M|$ may be exponentially larger than the syntactic description of M, which uses formulas for ϕ^i, ϕ^o.

Compatibility and composition. The basic principle is that two interfaces are compatible if they can be made to work together correctly. When two interfaces are composed, the outputs of one interface may be fed as inputs to the other interface. Thus, the possibility arises that the output behavior of one interface violates the input assumptions of the other. The two interfaces are called compatible if the environment can ensure that no such I/O violations occur. The assurance that the environment behaves in a way that avoids all I/O violations is, then, the input assumption of the composite interface. Formally, given two A/G interfaces M and N, define $V^o = V_M^o \cup V_N^o$ and $V^i = (V_M^i \cup V_N^i) \setminus V^o$. Let $Q = Q_M \times Q_N$ and $\hat{q} = (\hat{q}_M, \hat{q}_N)$. For all $(p, q), (p', q') \in Q_M \times Q_N$, let $\phi^o(p, q) = (\phi_M^o(p) \wedge \phi_M^o(q))$ and $\rho((p, q), (p', q')) = (\rho_M(p, p') \wedge \rho_N(q, q'))$. The interfaces M and N are *compatible* if (1) $V_M^o \cap V_N^o = \emptyset$ and

(2) there is a function ψ^i that assigns to all states $(p, q) \in Q$ a satisfiable predicate on V^i such that:

> For all initialized runs $(p_0, q_0), (v_0^i, v_0^o), (p_1, q_1), (v_1^i, v_1^o), \ldots$ of the A/G interface $\langle V^i, V^o, Q, \hat{q}, \psi^i, \phi^o, \rho \rangle$ and all $k \geq 0$, we have $(v_k^i \uplus v_k^o) \models (\phi_M^i(p_k) \wedge \phi_N^i(q_k))$. (†)

If M and N are compatible, then the *composition* $M \| N = \langle V^i, V^o, Q, \hat{q}, \phi^i, \phi^o, \rho \rangle$ is the A/G interface with the function ϕ^i that maps each state $(p, q) \in Q$ to a satisfiable predicate on V^i such that for all functions ψ^i that satisfy the condition (†), and all $(p, q) \in Q$, we have $\psi^i(p, q) \Rightarrow \phi^i(p, q)$; i.e., the input assumptions ϕ^i are the weakest conditions on the environment of the composite interface $M \| N$ which guarantee the input assumptions of both components. Algorithms for checking compatibility and computing the composition of A/G interfaces are given in [55]. These algorithms use the game representation of interfaces.

4.3 Resource Interfaces

Let $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\pm\infty\}$. A *resource algebra* is a tuple $A = \langle \mathbb{L}, \oplus, \Theta \rangle$ consisting of:

- A set $\mathbb{L}$ of *resource labels*, each signifying a level of consumption or production for a set of resources.
- A binary *composition operator* $\oplus$: $\mathbb{L}^2 \to \mathbb{L}$ on resource labels.
- A *value function* Θ: $\mathbb{L}^\omega \to \mathbb{Z}_\infty$, which assigns an integer value (or infinity) to every infinite sequence of resource labels.

A *resource interface* over A is a pair $R = (M, \lambda)$ consisting of an A/G interface $M = \langle \cdot, \cdot, Q, \hat{q}, \cdot, \cdot, \cdot \rangle$ and a labeling function λ: $Q \to \mathbb{L}$, which maps every state of

the interface to a resource label. The *size* of the resource interface is $|R| = |M| + \sum_{q \in Q} |\lambda(q)|$, where $|\ell|$ is the space required to represent the label $\ell \in \mathbb{L}$. The runs of R are the runs of M, etc. Given a run $\pi = q_0, (v_0^i, v_0^o), q_1, (v_1^i, v_1^o), \ldots$, we write $\lambda(\pi) = \lambda(q_0), \lambda(q_1), \ldots$ for the induced infinite sequence of resource labels. Given a state $q \in Q$, the *value at* q is the minimum value that player Input can achieve for the outcome of the game from q, irrespective of the moves chosen by player Output: $\mathit{val}(q) = \inf_{\sigma^i \in \Sigma^i} \sup_{\sigma^o \in \Sigma^o} \Theta(\lambda(\mathit{out}(q, \sigma^i, \sigma^o)))$. The state q is Δ-*compliant*, for $\Delta \in \mathbb{Z}_\infty$, if $\mathit{val}(q) \leq \Delta$. We write $Q_\Delta^{rc} \subseteq Q$ for the set of Δ-compliant states. The resource interface R is Δ-*compliant* if the initial state $\hat{q}$ is Δ-compliant, and the *value* of R is $\mathit{val}(\hat{q})$.

Given two resource interfaces $R = (M_R, \lambda_R)$ and $S = (M_S, \lambda_S)$ over the same resource algebra A, define λ: $Q_R \times Q_S \to \mathbb{L}$ such that $\lambda(p, q) = \lambda_R(p) \oplus \lambda_S(q)$. The resource interfaces R and S are Δ-*compatible*, for $\Delta \in \mathbb{Z}_\infty$, if (1) the underlying A/G interfaces M_R and M_S are compatible, and (2) the resource interface $(M_R \| M_S, \lambda)$ over A is Δ-compliant. Note that Δ-compatibility does not require that both component interfaces R and S are Δ-compliant. Indeed, if R consumes a resource produced by S, it may be the case that R is not Δ-compliant on its own, but is Δ-compliant when composed with S. This shows that different applications call for different definitions of composition for resource interfaces, and we refrain from a generic definition. We use, however, the abbreviation $R \| S = (M_R \| M_S, \lambda)$.

The class of resource interfaces over a resource algebra A is denoted $\mathcal{R}[A]$. We present four examples of resource algebras and the corresponding interfaces.

Pure node limit interfaces. The resource labels of a node limit interface specify, for each state q, an amount $\lambda(q) \in \mathbb{N}$ of resource usage in q (say, power consumption).

When the states of two interfaces are composed, their resource usage is additive. The number $\Delta \geq 0$ provides an upper bound on the amount of resource available at every state. A state q is Δ-compliant if player Input can ensure that, when starting from q, the resource usage never exceeds Δ. The value at q is the minimum amount Δ of resource that must be available at all states for q to be Δ-compliant. Formally, the *pure node limit algebra* A^t is the resource algebra with $\mathbb{L}^t = \mathbb{N}$ and $\oplus^t = +$ and $\Theta^t(n_0, n_1, \ldots) = \sup_{k \geq 0} n_k$. The resource interfaces in $\mathcal{R}[A^t]$ are called *pure node limit interfaces*. Throughout this chapter, we assume that all numbers, including the state labels $\lambda(q)$ of pure node limit interfaces as well as Δ, can be stored in space of a fixed size. It follows that the size of a pure node limit interface $R = (M, \lambda)$ is equal to the size of the underlying A/G interface M.

Example 4.1 Figure 4.1(a) shows the game associated with a pure node limit interface. For simplicity, the example is a turn-based game in which player Input makes moves in circle states, and player Output makes moves in square states. The numbers inside the states represent their resource labels. The solid arrows show the moves available to the players, and the dashed arrows indicate the optimal strategies for the two players. Note that at the initial state A, state E is a better choice than C for player Input in spite of having a greater resource label. It is easy to see that the value of the game (at A) is 15. ∎

Büchi node limit interfaces. While pure node limit interfaces ensure the safe usage of a resource with a node limit, they may allow some systems to never use the resource by not doing anything useful. To rule out this possibility, we may augment the pure node limit algebra with a generalized Büchi objective, which requires that certain state labels be visited infinitely often. A state q, then, is Δ-compliant if player

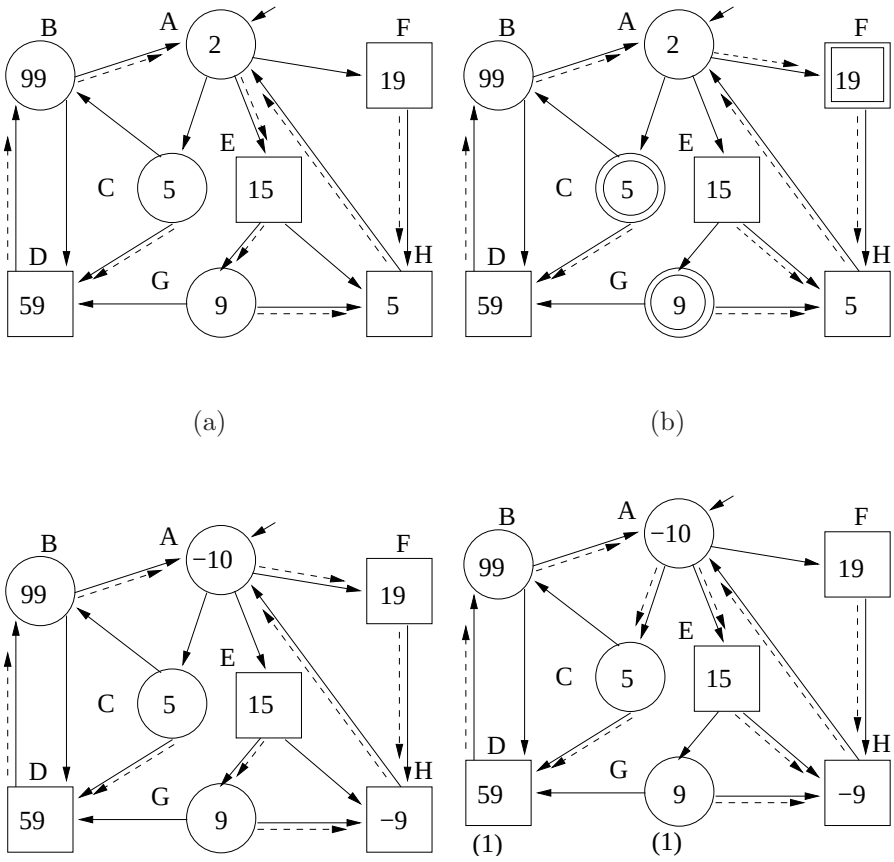


Figure 4.1. Games illustrating the four classes of resource interfaces.

Input can ensure that, when starting from q, the Büchi conditions are satisfied *and* resource usage never exceeds Δ. The formal definition of Büchi conditions within a resource algebra is somewhat technical. The *Büchi node limit algebra* A^{bt} is defined as follows, for a fixed set of labels $\mathcal{L}$:

- $\mathbb{L}^{bt}$ consists of triples $\langle n, \alpha, \beta \rangle \in \mathbb{N} \times 2^{\mathcal{L}} \times 2^{\mathcal{L}}$, where $n \in \mathbb{N}$ indicates the current level of resource usage, $\alpha \subseteq \mathcal{L}$ is a set of state labels that each need to be

repeated infinitely often, and $\beta \subseteq \mathcal{L}$ is the set of state labels that are satisfied in the current state.

- $\langle n, \alpha, \beta\rangle \oplus^{bt} \langle n', \alpha', \beta'\rangle = \langle n + n', \alpha \uplus \alpha', \beta \uplus \beta'\rangle$.
- We distinguish two cases, depending on whether or not the generalized Büchi objective is violated: $\Theta^{bt}(\langle n_0, \alpha_0, \beta_0\rangle, \langle n_1, \alpha_1, \beta_1\rangle, \ldots) = +\infty$ if there is an $\ell \in \alpha_0$ and a $k \geq 0$ such that for all $j \geq k$, we have $\ell \notin \beta_j$; otherwise, $\Theta^{bt}(\langle n_0, \alpha_0, \beta_0\rangle, \langle n_1, \alpha_1, \beta_1\rangle, \ldots) = \sup_{k\geq 0} n_k$.

The resource interfaces in $\mathcal{R}[A^{bt}]$ are called *Büchi node limit interfaces*. The *number of Büchi conditions* of a Büchi node limit interface $R = (M, \lambda)$ is $|\hat{\alpha}|$, where $\hat{\alpha}$ is the second component of the label $\lambda(\hat{q})$ for the initial state $\hat{q}$ of M.

Example 4.2 Figure 4.1(b) shows a Büchi node limit game with a single Büchi condition. The graph is the same as in Example 4.1. The states with double borders are Büchi states, i.e., one of them needs to be repeated infinitely often. Note that the optimal output strategy at E has changed, because C is a Büchi state but H is not. This forces player Input to prefer at A state F over E in order to satisfy the Büchi condition. The value of the game is now 19. ∎

Pure path limit interfaces. The resource labels of an path limit interface specify, for each state q, the amount of resource $\lambda(q) \in \mathbb{Z}$ that is produced (if $\lambda(q) > 0$) or consumed (if $\lambda(q) < 0$) at q. When the states of two interfaces are composed, their resource expenditures are added. The number $\Delta \geq 0$ provides the initial amount of the resource available. A state q is Δ-compliant if player Input can ensure that, when starting from q, the system can run forever without the available resource dropping below 0. The value at q is the minimum amount Δ of initial resource necessary for q to

be Δ-compliant. Formally, the *pure path limit algebra* A^e is the resource algebra with $\mathbb{L}^e = \mathbb{Z}$ and $\oplus^e = +$ and $\Theta^e(d_0, d_1, \ldots) = -\inf_{k \geq 0} \sum_{0 \leq j \leq k} d_j$. The resource interfaces in $\mathcal{R}[A^e]$ are called *pure path limit interfaces*. To characterize the complexity of the algorithms, we let the *maximal resource consumption* of a pure path limit interface $R = (M, \lambda)$ be 1 if $\lambda(q) \geq 0$ for all states $q \in Q$, and $-\min_{q \in Q} \lambda(q)$ otherwise.

Example 4.3 Figure 4.1(c) shows a pure path limit game. Player Input has a strategy to run forever when starting from the initial state A with 9 units of the resource, but 8 is not enough initial resource; thus the game has the value 9. ∎

Reward path limit interfaces. Some systems have the possibility of saving resource by doing nothing useful. To rule out this possibility, we may use a Büchi objective as in the case of node limit interfaces. For variety's sake, we provide a different approach. We label each state q not only with an resource expenditure, but also with a reward, which represents the amount of useful work performed by the system when visiting q. A reward path limit algebra specifies a minimum acceptable reward Λ. A state q, then, is Δ-compliant if player Input can ensure that, when starting from q with resource Δ, the reward Λ can be obtained without the available resource dropping below 0. For $\Lambda \in \mathbb{N}$, the Λ-*reward path limit algebra* A^{re}_Λ is defined as follows:

- $\mathbb{L}^{re} = \mathbb{Z} \times \mathbb{N}$. The first component of each label represents a resource expenditure; the second component represents a reward.
- $\langle d, n \rangle \oplus^{re} \langle d', n' \rangle = \langle d + d', n + n' \rangle$.
- There are two cases: $\Theta^{re}_\Lambda(\langle d_0, n_0 \rangle, \langle d_1, n_1 \rangle, \ldots) = +\infty$ if $\sum_{j \geq 0} n_j < \Lambda$; other-

wise, let $k^* = \min_{k \geq 0}(\sum_{0 \leq j \leq k} n_j \geq \Lambda)$ and define $\Theta^{re}_\Lambda(\langle d_0, n_0\rangle, \langle d_1, n_1\rangle, \ldots) = -\inf_{0 \leq k \leq k^*} \sum_{0 \leq j \leq k} d_j$.

The resource interfaces in $\mathcal{R}[A^{re}_\Lambda]$ are called Λ-*reward path limit interfaces*. The *maximal resource consumption* of a reward path limit interface is defined as for pure path limit interfaces, with the proviso that only the first components of resource labels are considered.

Example 4.4 Figure 4.1(d) shows a Λ-reward path limit game with $\Lambda = 1$. The numbers in parentheses represent rewards; states that are not labeled with parenthesized numbers have reward 0. The optimal choice of player Input at state A is E, precisely the opposite of the pure path limit case. If player Output chooses G at E, then the reward 1 is won, and player Input's objective is accomplished. If player Output instead chooses H at E, then 4 units of resource are gained in the cycle A,E,H,A. By pumping this cycle, player Input can gain sufficient resource to eventually choose the path A,C,D and win the reward 1. Hence the game has the value 5. Note that this example shows that reward path limit games may not have memoryless winning strategies. ∎

4.4 Algorithms

Let A be a resource algebra. We are interested in the following questions:

Verification Given two resource interfaces $R, S \in \mathcal{R}[A]$, and $\Delta \in \mathbb{Z}_\infty$, are R and S Δ-compatible?

Design Given two resource interfaces $R, S \in \mathcal{R}[A]$, for which values $\Delta \in \mathbb{Z}_\infty$ are R and S Δ-compatible?

To answer these questions, we first need to check the compatibility of the underlying A/G interfaces M_R and M_S. Then, for the qualitative verification question, we need to check if the resource interface $R \| S \in \mathcal{R}[A]$ is Δ-compliant, and for the quantitative design question, we need to compute the value of $R \| S$. Below, for $A \in \{A^t, A^{bt}, A^e, A^{re}\}$, we provide algorithms for checking if a given resource interface $R \in \mathcal{R}[A]$ is Δ-compliant, and for computing the value of R. We present the algorithms in terms of the game representation $G_R = \langle Q, \hat{q}, \gamma^i, \gamma^o, \delta \rangle$ of the interface. The algorithms have been implemented in our tool CHIC [1].

Pure node limit interfaces. For $n \in \mathbb{N}$, let $Q_{\leq n} = \{q \in Q \mid \lambda(q) \leq n\}$. For $\Delta \geq 0$, a pure node limit interface R is Δ-compliant iff player Input can win a game with the safety objective of staying forever in $Q_{\leq \Delta}$. Such safety games can be solved as usual using a *controllable predecessor* operator $CPre$: $2^Q \to 2^Q$, defined for all $X \subseteq Q$ by $CPre(X) = \{q \in Q \mid \exists v^i \in \gamma^i(q). \forall v^o \in \gamma^o(q). \delta(q, v^i, v^o) \in X\}$. The set of Δ-compliant states can then be written as the limit $Q^{rc}_\Delta = \lim_{k \to \infty} X_k$ of the sequence defined by $X_0 = Q$ and, for $k \geq 0$, by $X_{k+1} = Q_{\leq \Delta} \cap CPre(X_k)$. This algorithm can be written in μ-calculus notation as $Q^{rc}_\Delta = \nu X. (Q_{\leq \Delta} \cap CPre(X))$, where ν is the greatest fixpoint operator.

To compute the value of R, we propose the following algorithm. We introduce two mappings $lmax$: $2^Q \to \mathbb{N}$ and $below$: $2^Q \to 2^Q$. For $X \subseteq Q$, let $lmax(X) = \max\{\lambda(q) \mid q \in X\}$ be the maximum label of a state in X, and let $below(X) = \{q \in X \mid \lambda(q) < lmax(X)\}$ be the set of states with labels below the maximum. Then, define $X_0 = Q$ and, for $k \geq 0$, define $X_{k+1} = \nu X. (below(X_k) \cap CPre(X))$. For $k \geq 0$ and $q \in X_k \setminus X_{k+1}$, we have $val(q) = lmax(X_k)$.

While it may appear that computing the fixpoint $\nu X. (Q_{\leq \Delta} \cap CPre(X))$ requires

quadratic time (computing *CPre* is linear in $|R|$, and we need at most $|Q|$ iterations), this can be accomplished in linear time. The trick is to use a refined version of the algorithm, where each move pair $\langle v^i, v^o \rangle$ is considered at most once. First, we remove from the fixpoint all states q' such that $\lambda(q') > \Delta$. Whenever a state $q' \in Q$ is removed from the fixpoint, we propagate the removal backward, removing for all $q \in Q$ any move pair $\langle v^i, v^o \rangle \in \langle \gamma^i(q), \gamma^o(q) \rangle$ such that $\delta(q, v^i, v^o) = q'$ and, whenever $\langle v^i, v^o \rangle$ is removed, removing also $\langle v^i, \hat{v}^o \rangle$ for all $\hat{v}^o \in \gamma^o(q)$. The state q is itself removed if all its move pairs are removed. Once the removal propagation terminates, the states that have not been removed are precisely the Δ-compliant states. In order to implement efficiently the algorithm for computing the value of a node limit interface, we compute X_{k+1} from X_k by removing the states having the largest label, and then back-propagating the removal. In order to compute $\mathit{below}(X_k)$ efficiently for all k, we construct a list of states sorted according to their label.

Theorem 4.1 *Given a pure node limit interface R of size n, and $\Delta \in \mathbb{Z}_\infty$, we can check the Δ-compliance of R in time $O(n)$, and we can compute the value of R in time $O(n \cdot \log n)$.*

Büchi node limit interfaces. Given a Büchi node limit interface R, let $\lambda(\hat{q}) = \langle \hat{n}, \hat{\alpha}, \hat{\beta} \rangle$, $|\hat{\alpha}| = m$, and $\hat{\alpha} = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$. Let $B^i = \{q \in Q \mid \lambda(q) = \langle n^q, \alpha^q, \beta^q \rangle$ and $\alpha_i \in \beta^q\}$ be the i-th set in the generalized Büchi objective, for $1 \leq i \leq m$. We can compute the set of Δ-compliant states of R by adapting the fixpoint algorithm for solving Büchi games [64] as follows. Given two sets $Z, T \subseteq Q$ of states, we define $\mathit{Reach}(Z, T) \subseteq Q$ as the set of states from which player Input can force the game to T while staying in Z. Formally, define $\mathit{Reach}(Z, T) = \lim_{k \to \infty} W_k$, where $W_0 = \emptyset$ and $W_{k+1} = Z \cap (T \cup \mathit{CPre}(W_k))$ for $k \geq 0$. Then, for $Z \subseteq Q$ and $1 \leq$

$i \leq m$, we compute the sets $Y^i \subseteq Q$ as follows. Let $i' = (i \bmod m) + 1$ be the successor of i in the cyclic order $1, 2, \ldots, m, 1, \ldots$ Let $Y^i_0 = Q$, and for $j \geq 0$, let $Y^i_{j+1} = \mathit{Reach}(Z, B^i \cap \mathit{CPre}(Y^{i'}_j))$. Intuitively, the set Y^i_{j+1} consists of the states from which Input can, while staying in Z, first reach B^i and then go to $Y^{i'}_j$. For $1 \leq i \leq m$, let the fixpoint be $Y^i = \lim_{j\to\infty} Y^i_j$: from Y^i, Input can reach B^i while staying in Z; moreover, once at B^i, Input can proceed to $Y^{i'}$. Hence, Input can visit the sets $B^1, B^2, \ldots, B^m, B^1, \ldots$ cyclically, satisfying the generalized Büchi acceptance condition. Denoting by $\mathit{GBüchi}(Z, B^1, \ldots, B^m) = Y^1 \cup Y^2 \cup \ldots \cup Y^m$, we can write the set of Δ-compliant states of the interface as $Q^{rc}_\Delta = \mathit{GBüchi}(Q_{\leq\Delta}, B^1, \ldots, B^m)$.

The algorithm for computing the value of a Büchi node limit interface can be obtained by adapting the algorithm for Δ-compliance, similarly to the case for pure node limit interfaces. Let $X_0 = Q$, and for $k \geq 1$, let $X_{k+1} = \mathit{GBüchi}(\mathit{below}(X_k), B^1, \ldots, B^m)$. Then, for a state $q \in X_k \setminus X_{k+1}$, we have $\mathit{val}(q) = \mathit{lmax}(X_k)$.

Since the set $\mathit{Reach}(Z, T)$ can be computed in time $O(m \cdot |R|)$, using again a backward propagation procedure, the computation of the set of Δ-compliant states of the interface requires time $O(m \cdot |R|^2)$, in line with the complexity for solving Büchi games. The value of Büchi node limit games can also be computed in the same time. In fact, Y^i for iteration $k+1$ (denoted $Y^i(k+1)$) can be obtained from Y^i for k (denoted $Y^i(k)$) by $Y^i_0(k+1) = Y^i(k)$ and, for $j \geq 0$, by $Y^i_{j+1}(k+1) = \mathit{Reach}(X_k \cap Y^i(k), B^i \cap \mathit{CPre}(Y^{i'}_j(k+1)))$. We then have $Y^i(k+1) = \lim_{j\to\infty} Y^i_j(k+1)$. Hence, for $1 \leq i \leq m$, the sets $Y^i(0)$, $Y^i(1)$, $Y^i(2)$, ... can be computed by progressively removing states. As each removal (which requires the computation of *Reach*) is linear-time, the overall algorithm is quadratic. These considerations lead to the following theorem.

Theorem 4.2 *Given a Büchi node limit interface R of size n with m Büchi conditions, and $\Delta \in \mathbb{Z}_\infty$, we can check the Δ-compliance of R and compute its value in time $O(n^2 \cdot m)$.*

Pure path limit interfaces. Given a pure path limit interface R, the value at state $q \in Q$ is given by $val(q) = \inf_{\sigma^i \in \Sigma^i} \sup_{\sigma^o \in \Sigma^o} \{\Theta(\lambda(out(q, \sigma^i, \sigma^o)))\}$. To compute this value, we define an *path limit predecessor operator EPre*: $(Q \to \mathbb{Z}_\infty) \to (Q \to \mathbb{Z}_\infty)$, defined for all f: $Q \to \mathbb{Z}_\infty$ and $q \in Q$ by

$$EPre(f)(q) = -\lambda(q) + \max\{0, \min_{v^i \in \gamma^i(q)} \max_{v^o \in \gamma^o(q)} f(\delta(q, v^i, v^o))\}.$$

Intuitively, $EPre(f)(q)$ represents the minimum resource Input needs for performing one step from q without exhausting the resource, and then continuing with path limit requirement f. Consider the sequence of functions $f_0, f_1, \ldots$: $Q \to \mathbb{Z}_\infty$, where f_0 is the constant function such that $f_0(q) = -\infty$ for all $q \in Q$, and where $f_{k+1} = EPre(f_k)$ for $k \geq 0$. The functions in the sequence are pointwise increasing: for all $q \in Q$ and $k \geq 0$, we have $f_k(q) \leq f_{k+1}(q)$. Hence the limit $f_* = \lim_{k \to \infty} f_k$ (defined pointwise) always exists. From the definition of *EPre*, it can be shown by induction that $f_*(q) = val(q)$. The problem is that the sequence $f_0, f_1, \ldots$ may not converge to f_* in a finite number of iterations. For example, if the game has a state q with $\lambda(q) < 0$ and whose only transitions are self-loops, then $f_*(q) = +\infty$, but the sequence $f_0(q), f_1(q), \ldots$ never reaches $+\infty$. To compute the limit in finitely many iterations, we need a stopping criterion that allows us to distinguish between divergence to $+\infty$ and convergence to a finite value. The following lemma provides such a stopping criterion.

Lemma 4.1 *For all states q of a pure path limit interface, either $val(q) = +\infty$ or $val(q) \leq -\sum_{p \in Q} \min\{0, \lambda(p)\}$.*

This lemma is proved in a fashion similar to a theorem in [63], by relating the value of the path limit interface to the value along a loop in the game. Let $v^+ = -\sum_{p \in Q} \min\{0, \lambda(p)\}$. If $f_k(q) > v^+$ for some $k \geq 0$, we know that $f_*(q) = +\infty$. This suggests the definition of a modified operator $\mathit{ETPre}\colon (Q \to \mathbb{Z}_\infty) \to (Q \to \mathbb{Z}_\infty)$, defined for all $f\colon Q \to \mathbb{Z}_\infty$ and $q \in Q$ by

$$\mathit{ETPre}(f)(q) = \begin{cases} \mathit{EPre}(f)(q) & \text{if } \mathit{EPre}(f)(q) \leq v^+, \\ +\infty & \text{otherwise.} \end{cases}$$

We have $f_* = \lim_{k \to \infty} f_k$, where $f_0(q) = -\infty$ for all $q \in Q$, and $f_{k+1} = \mathit{ETPre}(f_k)$ for $k \geq 0$. Moreover, there is $k \in \mathbb{N}$ such that $f_k = f_{k+1}$, indicating that the limit can be computed in finitely many iterations. Once f_* has been computed, we have $\mathit{val}(q) = f_*(q)$ and $Q^{rc}_\Delta = \{q \in Q \mid f_*(q) \leq \Delta\}$.

Let ℓ be the maximal resource consumption of R. We have $v^+ \leq |Q| \cdot \ell$. Consider now the sequence $f_0, f_1, \ldots$ converging to f_*: for all $k \geq 0$, either $f_{k+1} = f_k$ (in which case $f_* = f_k$ and the computation terminates), or there must be $q \in Q$ such that $f_k(q) < f_{k+1}(q)$. Thus, the limit is reached in at most $v^+ \cdot |Q| \leq |Q|^2 \cdot \ell$ iterations. Each iteration involves the evaluation of the *ETPre* operator, which requires time linear in $|R|$. This leads to the following result.

Theorem 4.3 *Given a pure path limit interface R of size n with maximal resource consumption ℓ, and $\Delta \in \mathbb{Z}_\infty$, we can check the Δ-compliance of R and compute its value in time $O(n^3 \cdot \ell)$.*

Reward path limit interfaces. Given a Λ-reward path limit interface R and $\Delta \in \mathbb{Z}$, to compute Q^{rc}_Δ and *val*, we use a dynamic programming approach reminiscent of that used in the solution of shortest-path games [69]. We iterate over a set of *reward path-limit allocations* $\mathcal{E}\colon Q \to (\{0, \ldots, \Lambda\} \to \mathbb{Z}_\infty)$. Intuitively, for $f \in \mathcal{E}$, $q \in Q$,

and $r \in \{0, \ldots, \Lambda\}$, the value $f(q)(r)$ indicates the amount of resource necessary to achieve reward r before running out of the resource. For $e_1, e_2 \in \mathbb{Z}$, let $\mathrm{Mxe}(e_1, e_2) = \max\{e_1, e_2\}$ if $\max\{e_1, e_2\} \leq v^+$, and $\mathrm{Mxe}(e_1, e_2) = +\infty$ otherwise. For $r \in \mathbb{N}$, let $\mathrm{Mxr}(r) = \max\{0, r\}$. For $q \in Q$, use $\lambda(q) = \langle d(q), n(q) \rangle$. We define an operator *ERPre*: $\mathcal{E} \to \mathcal{E}$ on path-limit reward allocations by letting $g = \mathit{ERPre}(f)$, where $g \in \mathcal{E}$ is such that for all $q \in Q$ we have $g(q)(0) = 0$, and for all $r \in \{0, \ldots, \Lambda - 1\}$,

$$g(q)(r) = \mathrm{Mxe}(-d(q), -d(q) + \min_{v^i \in \gamma^i(q)} \max_{v^o \in \gamma^o(q)} f(\delta(q, v^i, v^o))(\mathrm{Mxr}(r - n(q)))).$$

Intuitively, given an path-limit reward allocation f, a state q, and a reward r, $\mathit{ERPre}(f)(q)(r)$ represents the minimum resource to achieve reward r from state q given that the next-state path-limit reward allocation is f. Let $f_0 \in \mathcal{E}$ be defined by $f_0(q)(r) = +\infty$, for $q \in Q$ and $r \in \{0, \ldots, \Lambda\}$, and for $k \geq 0$, let $f_{k+1} = \mathit{ERPre}(f_k)$. The limit $f_* = \lim_{k \to \infty} f_k$ (defined pointwise) exists; in fact, for all $q \in Q$ and $r \in \{0, \ldots, \Lambda\}$, we have $f_{k+1}(q)(r) \leq f_k(q)(r)$. For all $q \in Q$, we then have $\mathit{val}(q) = f_*(q)(\Lambda)$, and $q \in Q^{rc}_\Delta$ if $f_*(q)(\Lambda) \leq \Delta$.

The complexity of this algorithm can be characterized as follows. For all $q \in Q$, $r \in \{0, \ldots, \Lambda\}$, and $f \in \mathcal{E}$, the path limit $f(q)(r)$ can assume at most $1 + v^+ \leq 1 + \ell \cdot |Q|$ values, where ℓ is the maximal resource consumption in R. Since each of these values is monotonically decreasing, the limit f_* is computed in at most $O(|Q|^2 \cdot \ell \cdot \Lambda)$ iterations. Each iteration has cost $|R| \cdot \Lambda$.

Theorem 4.4 *Given a Λ-reward path limit interface R of size n with maximal resource consumption ℓ, and $\Delta \in \mathbb{Z}_\infty$, we can check the Δ-compliance of R and compute its value in time $O(n^3 \cdot \ell \cdot \Lambda^2)$.*

4.5 Examples

We sketch two small case studies that illustrate how resource interfaces can be used to analyze resource-constrained systems.

4.5.1 Distribution of resources in a Lego robot system

We use resource interfaces to analyze the schedulability of a Lego robot control program comprising several parallel threads. In this setup, player Input is a "resource broker" who distributes the resources among the threads. The system is compatible if Input can ensure that all resource constraints are met.

The Lego robot. We have programmed a Lego robot that must execute various commands received from a base station through infrared (ir) communication, as well as recover from bumping into obstacles. Its software is organized in 5 parallel threads, interacting via a central repository. The thread Scan Sensors (S) scans the values of the sensors and puts these into the repository, Motion (M) executes the tasks from the base station, Bump Recovery (B) is executed when the robot bumps into an object, Telemetry (T) is responsible for communication with the base station and the Goal Manager (G) manages the various goals. There are 3 mutex resources: the motor (m), the ir sensor (s) and the central repository (c). Furthermore, energy is consumed by the motor and ir sensor. We model each thread as a resource interface; our model is open, as more threads can be added later.

Checking schedulability using pure node limit interfaces. First, we disregard the energy consumption and consider the question whether all the mutex requirements

can be met. To this end, we model each thread $i \in \{S, M, B, T, G\}$ as a node limit interface (M^i, λ^i) with node limit value $\Delta = 1$. The resource labeling $\lambda^i = (\lambda^i_m, \lambda^i_c, \lambda^i_s)$ is such that $\lambda^i_R(q)$ indicates whether, in state q, thread i owns resource R. The underlying A/G interface M_i has, for each resource $R \in \{m, c, s\}$, a boolean input variable gr^i_R (abbreviated R in the figures) indicating whether Input grants R to i. We also model a resource interface (M^E, λ^E) for the environment, expressing that bumps do not occur too often. This interface does not use any resources, i.e. $\lambda^E_r(q) = 0$ for all states q and all resources R. These resource interfaces are 1-compatible iff all mutex requirements are met.[1]

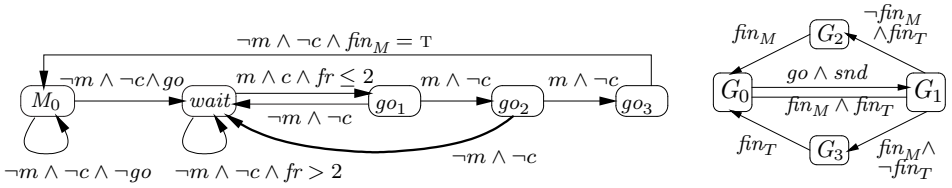
Figure 4.2 presents the A/G interfaces for Motion and Goal Manager; the others are modeled in a similar fashion. Also, rather than with $\rho(p, q)$, we label the edges $\rho(p, q) \wedge \phi^i(p) \wedge \phi^o(q)$. The tread Motion in Figure 4.2(a) has one boolean output variable fin_M, indicating whether it has finished a command from the base station. Besides the input variables gr^M_m, gr^M_c and gr^M_s discussed above, Motion has an input variable *fr* controlled by Scan Sensors that counts the steps since the last scanning of the sensors. In the initial location M_0, Motion waits for a command *go* from the Goal Manager. Its input assumption is $\neg m \wedge \neg c$, indicating that Motion needs neither the motor nor the repository. When receiving a command, Motion moves to the location *wait*, where it tries to get hold of the motor and of the repository. Since Motion needs fresh sensor values, it requires $fr \leq 2$ to move on the next location; otherwise it does not need either resource. In the locations go_1, go_2 and go_3, Motion executes the command. It needs the motor and repository in go_1, and the motor only in go_2 and go_3. If, in locations go_1 or go_2, the motor is retrieved from Motion

[1]Note that the resource compliance of (Büchi) node limit games with multiple resource labelings can be checked along the same lines as the resource compliance of node limit games with single resource labelings.

(input $\neg m \wedge \neg c$, typically if Bump Recovery needs the motor), the thread goes back to location *wait*. When leaving location go_3, Motion sets $fin_M = \text{T}$, indicating the completion of a command. We let $fin_M = \text{F}$ on all other transitions. The labeling λ_r^M for $r \in R$ is given by: $\lambda_m^M(go_1) = \lambda_m^M(go_2) = \lambda_m^M(go_3) = \lambda_c^M(go_1)$. $\lambda_r^M(q) = 0$ in all other cases. (Note that $\lambda_R^i(q)$ is derivable from gr_R^i by considering edges leading to q.) The interface for *Goal Manager* (Figure 4.2(b)) has output variables *go* and *snd* through which it starts up the threads Motion and Telemetry in location G_0 and then waits for them to be finished. It does not use any resources.

Checking schedulability using Büchi node limit interfaces. The node limit interfaces before express safety, but not liveness: the resource broker is not forced to ever grant the motor to Motion or Telemetry, in which case they stay forever in the locations *wait* or $wait_1$ respectively. To enforce the progress of the threads, we add a Büchi condition expressing that the locations G_0 should be visited infinitely often. Thus, each state is a state label and we define the location labeling of thread G by $\kappa^G(q) = (\lambda^G(q), \{G_0\}, \{q\})$ and for $i \in \{S, M, B, T, E\}$ by $\kappa^i(q) = (\lambda^i(q), \emptyset, \emptyset)$, where λ^i is as before. Then all mutex requirements can be met, with the state G_0 being visited infinitely often, iff the resource interfaces are 1-compatible.

Analyzing energy consumption using reward path limit interfaces. Energy is consumed by the motor and the ir sensor. We define the energy expense for thread i at state q as $\lambda_e^i(q) = 5\lambda_m^i(q) + 2\lambda_s^i(q)$, expressing that the motor uses 5 energy units and the ir sensor 2. Currently, the system will always run out of energy because it is never recharged, but it is easy to add an interface for that. To prevent the system from saving energy by doing nothing at all, we specify a reward. A naive attempt would be to assign the reward to each location in each thread and sum the rewards



(a) Motion. (b) Goal manager.

Figure 4.2. A/G interfaces modeling a Lego robot.

upon composition. However, suppose that the reward acquired per energy unit is higher when executing Motion than when executing Telemetry. Then, the highest reward is obtained by always executing Motion and never doing Telemetry. This phenomenon is not a deficit of the theory, it is inherent when managing various goals. Since the latter is exactly the task of the goal manager, we reward the completion of a round of the goal manager. That is, we put $\lambda_r^G(G_0) = 1$ and $\lambda_r^i(q) = 0$ in all other cases. Then all mutex requirements can be met, while the system never runs out of power, iff the node limit interfaces (as defined before) are 1-compatible and their composition is 0-compliant as an path-limit reward interface.

4.5.2 Resource accounting for the PicoRadio network layer

The PicoRadio [50] project aims to create large-scale, self-organizing sensor networks using very low-power, battery-operated *piconodes* that communicate over wireless links. In these networks, it is not feasible to connect each node individually to a power line or to periodically change its battery. Energy-aware routing [129] strategies have been found necessary to optimize use of scarce energy resources.We show how our methodology can be profitably applied to evaluate networks and synthesize optimal routing algorithms.

A PicoRadio network. A *piconet* consists of a set of piconodes that can create, store, or forward packets using multi-hop routing. The piconet *topology* describes the position, maximal packet-creation rate, and packet-buffer capacity at each piconode, and capacity of each link. Each packet has a *destination*, which is a node in the network. A *configuration* of the network represents the number of packets of each destination currently stored in the buffer of each piconode. A configuration that assigns more packets to a node than its buffer size is not *legal*.The network moves from one configuration to another in a *round*. We assume that a piconode always uses its peak transmission capacity on an outgoing link as long as it has enough packets to forward on that link. Wireless transmission costs energy. Each piconode starts with an initial amount of energy, and can possibly gain energy by scavenging.

We are given a piconet with known topology and initial energy levels at each piconode. We wish to find a routing algorithm that makes the network satisfy a certain safety property, e.g., that buffer overflows do not occur (or that whenever a node has a packet to forward, it has enough energy to do so). A piconet together with such a property represents a concurrent finite-state safety game between player Packet Generator, and player Router. Each legal configuration of the network is represented by a game state; the state ERROR represents all illegal configurations. The guarded state transitions reflect the configuration changes the network undergoes from round to round as the players concurrently make packet creation and routing choices under the constraints imposed by the network topology. The state ERROR has a self-loop with guard TRUE and no outgoing transitions. The initial state corresponds to the network configuration that assigns 0 packets to each node. The winning condition is derived from the property the network must satisfy. If player Router has a winning strategy σ, a routing algorithm that makes the network satisfy the given property

under the constraints imposed by the topology exists and can be found from σ; else no such routing algorithm exists. We present several examples.

Finding a routing strategy to prevent buffer overflows. Let $\lambda(q_c) = 0$ for each state q_c that represents a legal configuration c, and let $\lambda(\textsc{error}) = 1$. If the pure node limit interface thus constructed is Δ-compliant for $\Delta = 0$, then a routing algorithm that prevents buffer overflow exists and can be synthesized from a winning strategy for player Router.

Finding the optimal buffer size for a given topology. We wish to find out the smallest buffer capacity (less than a given bound) each piconode must have so that there exists a routing algorithm that prevents buffer overflows. Let $\lambda(q_c) = \max_i \sum_j c_{ij}$ for all nodes i and packet destinations j, where c_{ij} is the number of packets with destination j in node i in configuration c. The value of the pure node limit interface thus constructed gives the required smallest buffer size.

Checking if the network runs forever using path limit interfaces. We wish to find if there exists a routing algorithm A_f that enables a piconet to run forever, assuming each piconode starts with energy e. Let e_{sc} be the energy scavenged by a piconode in each round. Let $\lambda(q_c) = e_{sc} - \max_i \sum_j (p \cdot \min(c_{ij}, l_i(r_i(j))))$, where q_c, i, j, and c_{ij} are as above, p is the energy spent to transmit a packet, $l_i(x)$ is the capacity of the link from node i to node x, and r_i is the routing table at node i; and $\lambda(\textsc{error}) = -1$. If the pure path limit interface thus constructed is Δ-compliant for $\Delta = e$, then A_f exists and is given by the Router strategy.

Finding the minimum energy required to achieve a given lifetime. We wish to find the minimum initial energy e such that there exists a routing algorithm A_r that makes each piconode run for at least r rounds. Let $\lambda(q_c) = (e_c, 1)$, where e_c is the energy label of configuration q_c defined in the pure path limit interface above, and 1 is a reward; and $\lambda(\text{ERROR}) = (-1, 0)$. For $\Lambda = r$, the value of the Λ-reward path-limit interface thus constructed gives e, and the Router strategy gives A_r.

Acknowledgements

The work reported in this chapter was conducted jointly with Prof Luca de Alfaro, Prof Thomas A. Henzinger, and Prof Mariëlle Stoelinga. This research was funded by Prof Henzinger supported in part by the DARPA grant F33615-00-C-1693, the MARCO grant 98-DT-660, the ONR grant N00014-02-1-0671, and the NSF grants CCR-0085949, CCR-0132780, CCR-0234690, and CCR-9988172. This chapter is based on a paper [42] presented at EMSOFT 2003, copyright held by Springer-Verlag Berlin Heidelberg[2], 2003.

[2] `http://www.springerlink.com`

Chapter 5

A Natural Extension of Automata

We define and study a quantitative generalization of the traditional boolean framework of model-based specification and verification. In our setting, propositions have integer values at states, and properties have integer values on traces. For example, the value of a quantitative proposition at a state may represent power consumed at the state, and the value of a quantitative property on a trace may represent energy used along the trace. The value of a quantitative property at a state, then, is the maximum (or minimum) value achievable over all possible traces from the state. In this framework, model checking can be used to compute, for example, the minimum battery capacity necessary for achieving a given objective, or the maximal achievable lifetime of a system with a given initial battery capacity. In the case of open systems, these problems require the solution of games with integer values.

Quantitative model checking and game solving is undecidable, except if bounds on the computation can be found. Indeed, many interesting quantitative properties, like minimal necessary battery capacity and maximal achievable lifetime, can be naturally specified by *quantitative-bound automata*, which are finite automata with

integer registers whose analysis is constrained by a bound function f that maps each system K to an integer $f(K)$. Along with the linear-time, automaton-based view of quantitative verification, we present a corresponding branching-time view based on a quantitative-bound μ-calculus, and we study the relationship, expressive power, and complexity of both views.

5.1 Introduction

Traditional algorithmic methods for the verification of finite-state systems, with a set P of *boolean* propositions, translate a system into a transition graph in which each vertex corresponds to a state of the system and is labeled by the propositions that hold in the state. A property of the system is specified by a temporal-logic formula over P or by an automaton over the alphabet 2^P. When the system is closed (i.e., its behavior does not depend on the environment), verification is reduced to *model checking* [46]; for open systems, verification requires *game solving* [18]. While successful for verifying hardware designs [34] and communication protocols [95], this approach cannot adequately handle infinite-state systems that arise, for example, in general software verification. Much research has therefore focused on infinite-state extensions, such as models whose vertices carry a finite, but unbounded amount of information, e.g., a pushdown store, or integer-valued registers [99]. Much of the reasoning about such systems, however, has still focused on boolean specifications (such as "is the buffer size always bounded by 5?") rather than answering quantitative questions (e.g., "what is the maximal buffer size?"). Moreover, the main challenge in most infinite-state formalisms has been to obtain decidability for checking boolean properties, usually by limiting the expressive power of the models or properties.

In contrast, the solution of *quantitative* questions, such as system power requirements and system lifetime, has been considered on a property-by-property basis. Often the solution consists, however, of two basic steps: first, a suitable system of constraints is set up whose solution gives the intended quantitative answer (a "dynamic program"); and second, by considering the characteristics of the system (number of states or maximal initial battery power), a bound is provided on the number of iterations required to solve the dynamic program. We systematize this ad-hoc approach to answering quantitative questions about infinite-state systems in order to make it accessible to design engineers. For this purpose, we extend the traditional boolean verification framework to an *integer-based* framework, which due to its generality permits the modeling of a wide variety of quantitative aspects and properties of systems [42, 31].[1] In particular, we generalize traditional boolean specification formalisms such as automata to the integer-based framework, so that an engineer can express the desired quantitative properties in a natural way. These quantitative automata are then automatically translated into dynamic programs for model checking and game solving. Finally, from parametric bounds given by the engineer, such as bounds on the value of a quantity or on the number of automaton steps necessary for computing a property, we automatically derive iteration bounds on solving the corresponding dynamic program. In all the examples we study, such as maximal lifetime of a system with given initial battery capacity, our generic, systematic approach matches the best known previous, property-specific algorithms.

Specifically, the models we consider, *quantitative structures*, are graphs with

[1] It should be noted that we use the term *quantitative*, as in quantitative verification, quantitative property, or quantitative μ-calculus, simply as referring to "integer-based" rather than "boolean." This is not to be confused with some literature, where the term *quantitative* is used to refer to "probabilistic" systems, and real values are obtained as results of evaluating boolean specifications [29, 97, 113, 58].

finitely many vertices, but every vertex is labeled by a set of *quantitative propositions*, each taking an integer value. For example, the label at each vertex may represent the amount of power consumed when the vertex is visited, or it may represent a buffer size, a time delay, a resource requirement, a reward, a cost, etc. The properties we check are quantitative properties of infinite paths, each representing a run of the system. For instance, we may ask for the peak power consumption along a path, or for the lifetime of a battery along the path given a certain amount of initial battery power (i.e., the number of transitions along the path until the initial battery power is used up). Such properties can be specified by an extension of traditional automata. While a traditional automaton maps infinite paths of a graph with boolean propositions (i.e., infinite words over the alphabet 2^P) to "accept" or "reject", we define *quantitative* automata, which map each infinite path of a graph with quantitative propositions (i.e., infinite words over the alphabet $\mathbb{N}^P$) to an integer. For example, if the proposition $p \in P$ describes the amount of power consumed when the current input letter is read, then an automaton specifying battery lifetime, given initial power $a \in \mathbb{N}$, maps each word $o_1 o_2 o_3 \ldots$ to the maximal $k \geq 0$ for which $\sum_{i=1}^{k} o_i(p)$ is at most a. In model checking, boolean properties of infinite paths can be interpreted either in an existential or universal way, asking whether the property is true on some or all paths from a given state. In quantitative verification, we ask for the *maximal* or *minimal* value of a property over all paths from a state. For the battery lifetime property, this amounts to computing the maximal or minimal achievable lifetime (note that this corresponds to the battery lifetime in the cases that a scheduler resolves all nondeterminism in a friendly vs. an adversarial manner). In a game, where two players (system components) decide which path is taken, boolean properties are interpreted in an $\exists\forall$ fashion ("does player 1 have a strategy so that for all player 2 strategies the property is satisfied?"). Accordingly, we interpret quantitative properties in a *max*

min fashion ("what is the maximal value of the property that player 1 can achieve no matter how player 2 plays?").

Since quantitative automata subsume counter machines, model checking and game solving are undecidable. However, unlike much previous work on infinite-state verification, we do not focus on defining decidable subclasses, but we note that in many examples that arise from verification applications, it is often easy and natural to give a *bound function*. This function specifies, for given system parameters (such as number of states, maximal constants, etc.), a threshold when it is safe to conclude that the value of a quantitative property tends to infinity. Accordingly, we specify a quantitative property as a *quantitative-bound automaton*, which is a pair consisting of a quantitative automaton and a bound function. Note that bounds are not constant but depend on the size of the structure over which a specification is interpreted; they are *functions*. We consider *value-bound* functions, which constrain the maximal value of an automaton register, and *iteration-bound* functions, which constrain the maximal number of automaton transitions that need to be analyzed in order to compute the value of the property specified by the automaton. Iteration bounds directly give termination bounds for dynamic programs, and thus better iteration bounds yield faster verification algorithms. In particular, for the battery lifetime property, the generic dynamic-programming algorithms based on iteration bounds are more efficient than the finite-state algorithms derived from value bounds, and they match the best known algorithms that have been devised specifically for the battery lifetime property [42]. Given a value-bound function f, we can always obtain a corresponding iteration-bound function g: for quantitative automata with $|Q|$ control locations and k registers, and quantitative structures G, the iteration bound $g(G) = O(|Q| \cdot |G| \cdot f(G)^k)$ is sufficient and necessary. Moreover, for certain subclasses

of quantitative automata it is possible to derive better iteration bounds. For instance, for *monotonic* quantitative-bound automata (without decreasing register values), we derive iteration-bound functions that are linear with respect to given value-bound functions.

The verification problems for properties specified by quantitative-bound automata are finite-state, and therefore decidable. However, instead of reducing these problems to boolean problems, we provide algorithms that are based on generic and natural, integer-based dynamic programming formulations, where the bound function gives a termination guarantee for the evaluation of the dynamic program. We expect these algorithms to perform well in practice, as they (1) avoid artificial boolean encodings of integers and (2) match, in all the examples we consider, the complexity of the best known property-specific algorithms. The use of bound functions can be viewed as a generalization of bounded model checking [30] from the boolean to the quantitative case. In bounded model checking, the engineer provides a bound on the number of execution steps of a system along with a property. However, the bound is usually a constant independent of the structure, whereas our bound functions capture when search can be terminated without losing information about the structure. Therefore, in bounded model checking, only the structure diameter constitutes a bound function in our sense, because smaller bounds may give counterexamples but not proofs. Of course, as in bounded model checking, our approach could be used to quickly find counterexamples for quantitative verification problems even if the bound function gives values that are smaller than necessary for proof.

Quantitative automata specify dynamic programs. There is a second natural way to specify iterative computation: through the μ-calculus [103]. In a quantitative extension of the μ-calculus, each formula induces a mapping from vertices to integers,

and bound functions naturally specify a bound on the number of iterations for evaluating fixpoint expressions. More precisely, for a μ-formula φ, an iteration-bound function g specifies that if, during the iterative calculation of the value of a fixpoint expression in φ on a structure G, a stable value is not reached within $g(G)$ iterations, then the value is infinity. While quantitative extensions of the μ-calculus [97, 113, 58] have been defined before, they were interpreted over probabilistic structures and gave no iteration bounds. Finally, we give a translation from linear-time quantitative-bound automata to the branching-time quantitative-bound μ-calculus. For the purpose of game solving, as in the boolean case, the translation requires that the automaton is deterministic. This gives us symbolic algorithms for the quantitative verification of closed and open systems. Moreover, we show that the relationship [56] between boolean μ-formulas over *transition* graphs and boolean μ-formulas over *game* graphs carries over to the quantitative setting: a quantitative-bound μ-formula computes a particular quantitative property over two-player game graphs iff the formula computes the property over both existential and universal transition graphs (i.e., game graphs where one of the two players has no choices). This shows that the same integer-based symbolic iteration schemes can be used for verifying a quantitative property over both closed and open systems, provided the single-step operation is modified appropriately; this was previously known only for boolean structures, where the dynamic programs are degenerate [56].

5.2 The Integer-based Quantitative Setting

Quantitative properties. Let P be a nonempty, finite set of *quantitative propositions* (*propositions*, for short). A *quantitative observation* (*observation*, for short) is

a function o: $P \to \mathbb{N}$ mapping each proposition to a natural number (possibly 0). Let $\mathcal{O}$ be the set of observations. A *quantitative trace* (*trace*, for short) is an infinite sequence $w \in \mathcal{O}^\omega$ of observations. A *quantitative property* (*property*, for short) is a function π: $\mathcal{O}^\omega \to \mathbb{N} \cup \{\infty\}$ mapping each trace to a natural number or to infinity. Let Π denote the set of properties. These definitions generalize the boolean interpretation [46], where observations are maps from propositions to $\{0, 1\}$, and properties are maps from traces to $\{0, 1\}$. The following examples describe some quantitative properties.

Example 5.1 (Response time) Let $P = \{p\}$. Given $a \in \mathbb{N}$, the property rt_a: $\mathcal{O}^\omega \to \mathbb{N}$ maps each trace w to $rt_a(w) = \sup\{k \mid \exists w' \in \mathcal{O}^*, w'' \in \mathcal{O}^\omega \text{ such that } w = w' \cdot (p \mapsto a)^k \cdot w''\}$. Thus, $rt_a(w)$ is the supremal number of consecutive observations mapping the proposition p to the value a in the trace w. This may model the maximal time between a request and a response. The supremum may be infinity. This happens if $w = w' \cdot (p \mapsto a)^\omega$, or if for all $k \geq 0$, the trace w contains a subsequence with at least k successive observations mapping p to a (for example, p may be mapped to $abaabaaabaaaab\ldots$). ∎

Example 5.2 (Fair maximum) Let $P = \{p, q\}$. The property *fm*: $\mathcal{O}^\omega \to \mathbb{N}$ maps each trace w to the supremal value of the proposition p on w if the proposition q is nonzero infinitely often on w, and to 0 otherwise. The proposition q may model a fairness condition on traces [42]. Formally, $fm(o_0o_1o_2\ldots)$ is $\sup\{o_j(p) \mid j \geq 0\}$ if $\limsup\{o_j(q) \mid j \geq 0\} \neq 0$, and 0 otherwise. The supremum may be infinity. ∎

Example 5.3 (Lifetime) Let $P = \{p, c\}$. Given $a \in \mathbb{N}$, the property lt_a: $\mathcal{O}^\omega \to \mathbb{N}$ maps each trace $w = o_0o_1o_2\ldots$ to $lt_a(w) = \sup\{k \mid \sum_{j=0}^{k}(-1)^{c_j} \cdot o_j(p) \leq a\}$, where $c_j = 0$ if $o_j(c) = 0$, and $c_j = 1$ otherwise. Intuitively, if a zero (resp., nonzero) value

$o(c)$ denotes resource consumption (resp., resource gain) in a single step of $o(p)$ units, then $lt_a(w)$ is the supremal number of steps that can be executed without exhausting the resource, given a initial units of the resource. ∎

Example 5.4 (Peak running total) Let $P = \{p, c\}$ as in the previous example. The property *prt*: $\mathcal{O}^\omega \to \mathbb{N}$ maps each trace $w = o_0 o_1 o_2 \ldots$ to $prt(w) = \sup\{\sum_{j=0}^{k} (-1)^{c_j} \cdot o_j(p) \mid j \geq 0\}$, where again $c_j = 0$ if $o_j(c) = 0$, and $c_j = 1$ otherwise. Intuitively, if a resource is being consumed or gained over the trace w, then $prt(w)$ is the initial amount of the resource necessary so that the resource is never exhausted. ∎

Quantitative structures. A *quantitative system* (*system*, for short) is a tuple $K = (S, \delta, s_0, \langle\cdot\rangle)$, where S is a finite set of states, $\delta \subseteq S \times S$ is a total transition relation, $s_0 \in S$ is an initial state, and $\langle\cdot\rangle$: $S \to \mathcal{O}$ is an observation function that maps each state s to an observation $\langle s \rangle$. A two-player *quantitative game structure* (*game*, for short) is a tuple $G = (S, S_1, S_2, \delta, s_0, \langle\cdot\rangle)$, where S, δ, s_0, and $\langle\cdot\rangle$ are as in systems, and $S_1 \cup S_2 = S$ is a partition of the state space into player-1 states S_1 and player-2 states S_2. At player-1 states, the first player chooses a successor state; at player-2 states, the second player. Note that systems are special cases of games: if $S_i = S$, for $i \in \{1, 2\}$, then the game is called a *player-i system*. We use the term *structure* to refer to both systems and games.

A *trajectory* of the structure G is an infinite sequence $t = r_0 r_1 r_2 \ldots$ of states $r_j \in S$ such that the first state r_0 is the initial state s_0 of G, and $(r_j, r_{j+1}) \in \delta$ for all $j \geq 0$. The trajectory t induces the infinite sequence $\langle t \rangle = \langle r_0 \rangle \langle r_1 \rangle \langle r_2 \rangle \ldots$ of observations. A trace $w \in \mathcal{O}^\omega$ is *generated* by G if there is a trajectory t of G such that $w = \langle t \rangle$. A *player-i strategy*, for $i \in \{1, 2\}$, is a function ξ_i: $S^* \times S_i \to S$ that maps every

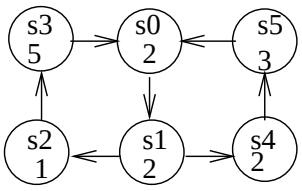


Figure 5.1. System K

nonempty, finite sequence of states to a successor of the last state in the sequence; that is, $(s, \xi_i(t, s)) \in \delta$ for every state sequence $t \in S^*$ and state $s \in S_i$. Intuitively, $\xi_i(t, s)$ indicates the choice taken by player i according to strategy ξ_i if the current state of the game is s, and the history of the game is t. We write Ξ_i for the set of player-i strategies. For two strategies $\xi_1 \in \Xi_1$ and $\xi_2 \in \Xi_2$, the *outcome* t_{ξ_1,ξ_2} of ξ_1 and ξ_2 is a trajectory of G, namely, $t_{\xi_1,\xi_2} = r_0 r_1 r_2 \ldots$ such that $r_0 = s_0$ and for all $j \geq 0$ and $i \in \{1, 2\}$, if $r_j \in S_i$, then $r_{j+1} = \xi_i(r_0 r_1 \ldots r_{j-1}, r_j)$.

Consider the system K shown in Figure 5.1, with the initial state s_0. Each state s_i of K is labeled with the value $\langle s_i \rangle(p)$ for a proposition p. Consider the property rt_2 from Example 5.1. For all traces w that correspond to trajectories of K of the form $(s_0 s_1 s_2 s_3)^*$, we have $rt_2(w) = 2$. For all traces w that correspond to trajectories of the form $(s_0 s_1 s_4 s_5)^*$, we have $rt_2(w) = 3$. Moreover, $rt_2(w) \leq 3$ for all traces w generated by K. Now consider a game played on the same structure K, where the state s_1 is a player-2 state. Consider the property lt_{14} from Example 3, supposing that $\langle s \rangle(c) = 0$ for all states s of K. The goal of player 2 is to maximize lifetime given initially 14 units of the resource. Consider the strategy where player 2 chooses s_4 at the first visit to s_1, and chooses s_2 thereafter. This strategy generates a trace w along which p is mapped to $2223\,(2215)^\omega$; hence $lt_{14}(w) = 7$. Note that all memoryless (i.e., history-independent) strategies lead to smaller lifetimes.

5.3 Quantitative-Bound Automata

5.3.1 Specifying Quantitative Properties

Syntax. We specify properties using automata. Let $\mathcal{O}$ be a given finite set of observations. Quantitative automata run over input traces in $\mathcal{O}^\omega$. The configuration of a quantitative automaton consists of a control location and an array of registers with values in $\mathbb{N}$. The transitions of quantitative automata are guarded by conditions on the values of the registers and the input observation, and involve, in addition to an update of the control location, also an update of the register values. A k-register *update function* is a recursive function u: $\mathbb{N}^k \times \mathcal{O} \rightharpoonup \mathbb{N}^k$ which may be partial. Let U denote the set of update functions. A *quantitative automaton* (*automaton*, for short) is a tuple $A = \langle Q, k, q_0, \gamma \rangle$, where Q is a finite set of control locations, $k \in \mathbb{N}$ is a number of registers, $q_0 \in Q$ is an initial location, and γ: $Q \to 2^{U \times Q}$ is a transition function that maps each location q to a finite set $\gamma(q)$ of pairs consisting of an update function and a successor location. We require that the transition function γ defines a total relation, namely, for each location $q \in Q$, each observation $o \in \mathcal{O}$, and all register values $\vec{x} \in \mathbb{N}^k$, there exists $(u, q') \in \gamma(q)$ such that $u(\vec{x}, o)$ is defined. For technical convenience, we furthermore assume that the automaton has a sink location $q_{halt} \in Q$: if the current location is q_{halt}, then for all observations, the next location is q_{halt} and the values of the registers remain unchanged; that is, $\gamma(q_{halt}) = \{(\lambda \vec{x}.\, \lambda o.\, \vec{x}, q_{halt})\}$. We write R for the array of registers, and $R[i] \in \mathbb{N}$ for the value of the i-th register, for $0 \leq i < k$.

Semantics. A *configuration* of the automaton A is a tuple $(q, v_0, v_1, \ldots, v_{k-1}) \in Q \times \mathbb{N}^k$ that specifies the current control location and the values of the registers. The initial configuration of the automaton is $c_{init} = (q_0, 0, 0, \ldots, 0)$, where all k registers

are initialized to 0. For an input $o \in \mathcal{O}$, the configuration $c' = (q', v'_0, v'_1, \ldots, v'_{k-1})$ is an *o-successor* of the configuration $c = (q, v_0, v_1, \ldots, v_{k-1})$, denoted by $c \xrightarrow{o} c'$, if there is a transition $(u, q') \in \gamma(q)$ such that $u(v_0, v_1, \ldots, v_{k-1}, o) = (v'_0, v'_1, \ldots, v'_{k-1})$. A *run* of the automaton A over a trace $o_0 o_1 o_2 \ldots \in \mathcal{O}^\omega$ is an infinite sequence $c_0 c_1 c_2 \ldots$ of configurations such that $c_0 = c_{init}$, and $c_j \xrightarrow{o_j} c_{j+1}$ for all $j \geq 0$. The *value* of the run $r = c_0 c_1 c_2 \ldots$ is defined as $val_A(r) = \limsup\{R[0](c_j) \mid j \geq 0\}$, that is, the value of r is the maximal value of the register $R[0]$ which occurs infinitely often along r, if this maximum is bounded; and otherwise the value is infinity. In other words, $val_A(r) = \infty$ iff for all $k \geq 0$, the value of the register $R[0]$ is infinitely often greater than k.

An automaton is *monotonic* if along every run, the value of each register cannot decrease. An automaton is *deterministic* if for every configuration c and input $o \in \mathcal{O}$, there is exactly one o-successor of c. While a deterministic automaton has a single run over every input trace, in general an automaton may have several runs over a given trace, each with a possibly different value. According to the *nondeterministic* (or *existential*) interpretation of automata, the *value* of an automaton A over a trace w, denoted $val_A^{\text{nondet}}(w)$, is the supremal value of all runs of A over w. Formally, $val_A^{\text{nondet}}(w) = \sup\{val_A(r) \mid r \text{ is a run of } A \text{ with } \langle r \rangle = w\}$. An alternative is the *universal* interpretation of automata, where the *value* of A over a trace w, denoted $val_A^{\text{univ}}(w)$, is the infimal value of all runs of A over w; that is, $val_A^{\text{univ}}(w) = \inf\{val_A(w, r) \mid r \text{ is a run of } A \text{ with } \langle r \rangle = w\}$. Note that a deterministic automaton A can be viewed as both a nondeterministic and a universal automaton. The (nondeterministic) automaton A *specifies* (or *computes*) the property $\pi \in \Pi$ if for all traces $w \in \mathcal{O}^\omega$, we have $val_A^{\text{nondet}}(w) = \pi(w)$. This definition captures traditional

Büchi automata as a special case: keep one register $R[0]$, which is set to 1 whenever the automaton visits a Büchi accepting control location, and set to 0 otherwise.

Model checking and game solving. Let K be a quantitative system. For a quantitative automaton A, the *max-value* of K with respect to A, denoted $val_A^{\max}(K)$, is the supremal value of all traces generated by K, where we choose the nondeterministic (rather than the universal) interpretation of automata. Formally, $val_A^{\max}(K) = \sup\{val_A^{\text{nondet}}(w) \mid w \text{ is a trace generated by } K\}$. The *min-value* of K with respect to A, denoted $val_A^{\min}(K)$, is the infimal value of all traces generated by K; that is, $val_A^{\min}(K) = \inf\{val_A^{\text{nondet}}(w) \mid w \text{ is a trace generated by } K\}$. Now consider a game G. The value of a strategy pair $\xi_1 \in \Xi_1$ and $\xi_2 \in \Xi_2$ with respect to a *deterministic* automaton A is the value $val_A(\xi_1, \xi_2) = val_A(t_{\xi_1,\xi_2})$ of A over the outcome of the strategies ξ_1 and ξ_2. The *game-value* of G with respect to a deterministic automaton A, denoted $val_A^{\text{maxmin}}(G)$, is defined as $\sup_{\xi_1 \in \Xi_1} \inf_{\xi_2 \in \Xi_2} val_A(\xi_1, \xi_2)$. This is the supremal value of A that player-1 can achieve against all player-2 strategies. The symmetric definition is omitted for brevity.

Given a system K and an automaton A, the *quantitative model-checking problem* (*model checking*, for short) is to determine $val_A^{\max}(K)$ and $val_A^{\min}(K)$. Given a game G and a deterministic automaton A, the *quantitative game-solving problem* (*game solving*, for short) is to determine $val_A^{\text{maxmin}}(G)$. Since registers can contain arbitrary natural numbers, we can encode 2-counter machines as monotonic automata, and hence the model-checking and game-solving problems are undecidable.

5.3.2 Bound Functions for Automata

Quantitative-bound automata. In order to solve model-checking problems and games, we equip quantitative automata with bound functions. A *quantitative-bound automaton* (QBA) (A, f) consists of a quantitative automaton A and a recursive function $f\colon\ \mathcal{G} \to \mathbb{N}$, where $\mathcal{G}$ is the set of quantitative structures (systems and games). To compute a property on a structure G, a QBA works with a bound $f(G)$ that depends on G. The motivation is that for many properties, the designer can provide a bound on the maximal value of the automaton registers, or on the number of automaton transitions that need to be executed in order to compute the value of the property if the value is finite. We thus have two interpretations of the bound function f: the *value-bound* interpretation, where $f(G)$ is a bound on the register values, and the *iteration-bound* interpretation, where $f(G)$ is a bound on the automaton transitions.

We define the value of a QBA over a trace generated by a structure for the two possible interpretations. Given a QBA (A, f), a structure G, and a trace w generated by G, let r be a run of the automaton A over w. The value of $r = c_0c_1c_2\ldots$ over w for the value-bound interpretation, denoted $val_{\mathrm{vbound}(A,f)}(r)$, is defined as follows: if there are an index $j \in \mathbb{N}$ and a register $R[i]$, for $0 \leq i < k$, such that $R[i](c_j) > f(G)$, then $val_{\mathrm{vbound}(A,f)}(r) = \infty$; otherwise $val_{\mathrm{vbound}(A,f)}(r) = val_A(r)$. Intuitively, the value-bound interpretation maps every trace that causes some register to exceed the value bound at some point, to ∞. The value of the run r over w for the iteration-bound interpretation, denoted $val_{\mathrm{ibound}(A,f)}(r)$, is defined as follows: if for all $0 \leq i < k$, we have $\max\{R[i](c_j) \mid f(G) \leq j \leq 2 \cdot f(G)\} = \max\{R[i](c_j) \mid 2 \cdot f(G) \leq j \leq 3 \cdot f(G)\}$, then $val_{\mathrm{ibound}(A,f)}(r) = \max\{R[0](c_j) \mid f(G) \leq j \leq 2 \cdot f(G)\}$; otherwise $val_{\mathrm{ibound}(A,f)}(r) = \infty$. Intuitively, the iteration-bound interpretation checks if the

maximal values of all registers stabilize within the iteration bound, and maps a trace to ∞ if some maximal register value does not stabilize.

Given a QBA (A, f), a system K, a game G, a trace generated by K or G, and two interpretations bound $\in \{\text{vbound}, \text{ibound}\}$, we define the values $val^{\text{nondet}}_{\text{bound}(A,f)}(w)$, $val^{\max}_{\text{bound}(A,f)}(K)$, $val^{\min}_{\text{bound}(A,f)}(K)$, and $val^{\text{maxmin}}_{\text{bound}(A,f)}(G)$ analogous to the corresponding definitions in Section 5.3.1 using $val_{\text{bound}(A,f)}(r)$ instead of $val_A(r)$. The QBA (A, f) *specifies* (or *computes*) the property π on a structure G if for all traces w generated by G, we have $val^{\text{nondet}}_{\text{bound}(A,f)}(w) = \pi(w)$. The following examples illustrate the idea.[2]

Example 5.5 (Fair maximum) The following QBA (A, f) specifies the property *fm* from Example 5.2 on all structures G. There are two registers. The register $R[1]$ keeps track of the maximal value of proposition p seen so far. Whenever proposition q has a nonzero value, the value of $R[1]$ is copied to $R[0]$; otherwise $R[0]$ is set to zero. If q has a nonzero value infinitely often, then the maximal value of p occurs infinitely often in $R[0]$; otherwise from some point on, $R[0]$ contains the value 0. The bound function f is defined as follows: if G contains the maximal value Δ for p, then $f(G) = \Delta$ is a suitable value-bound function; if G has N states, then $f(G) = N$ is a suitable iteration-bound function. ∎

Example 5.6 (Lifetime) The property lt_a from Example 5.3 can be computed on all structures G by the following QBA (A, f). Let $A = \langle\{q_0, q_{halt}\}, 2, q_0, \gamma\rangle$, where for all inputs $o \in \mathcal{O}$, we have $\gamma(q_0) = \{(o(c) \neq 0 \wedge R'[0] = R[0] + 1 \wedge R'[1] = R[1] - o(p), q_0), (o(c) = 0 \wedge R[1] + o(p) \leq a \wedge R'[0] = R[0] + 1 \wedge R'[1] = R[1] + o(p), q_0), (o(c) = 0 \wedge R[1] + o(p) > a \wedge R'[0] = R[0] \wedge R'[1] = R[1], q_{halt})\}$. In register

[2] In the examples, we write update functions as relations $u(\vec{x}, o, \vec{x}')$, where unprimed variables denote the values of variables before the update, and primed variables denote the values after the update.

$R[0]$ the automaton stores the number of transitions already taken, and in $R[1]$ it tracks the amount of the resource used so far; it continues to make transitions as long as it has a sufficient amount of the resource. If G contains N states and the maximal value Δ for p, then $f(G) = a + (N+1) \cdot \Delta$ is a suitable value-bound function, and $f(G) = N \cdot a + N \cdot (N+1) \cdot \Delta$ is a suitable iteration-bound function. ∎

5.3.3 Quantitative-Bound Model Checking and Game Solving

Given a system K and a QBA (A, f), the *quantitative-bound model-checking* problem is to determine $val^m_{\text{bound}(A,f)}(K)$, where bound $\in \{\text{vbound}, \text{ibound}\}$ and $m \in \{\max, \min\}$. Similarly, given a game G and a deterministic QBA (A, f), the problem of *solving quantitative-bound games* is to determine $val^{\text{maxmin}}_{\text{bound}(A,f)}(G)$, for bound $\in \{\text{vbound}, \text{ibound}\}$. Quantitative-bound model checking and game solving are decidable. In the case of value bounds, the state space is bounded by $O(|G| \cdot |Q| \cdot (f(G)+2)^k)$, where $|Q|$ is the size of the automaton with k registers, $|G|$ is the size of the structure, and f is the value-bound function. Let G be a structure such that for all propositions $p \in P$ and states $s \in S$, we have $\langle s \rangle(p) \leq \Delta$. Let C_0 be the maximal constant that appears syntactically in the description of the automaton A, and let $C_1 = f(G)$. Call $B = \max\{\Delta, C_0, C_1\}$ the *oblivion bound* for the QBA (A, f) and structure G. Let $g(G) = |G| \cdot |Q| \cdot (B+2)^k$, where A has k registers. Then $val^m_{\text{vbound}(A,f)}(G) = val^m_{\text{ibound}(A,g)}(G)$, for $m \in \{\max, \min, \text{maxmin}\}$. Thus, we can derive an iteration bound from a value bound.

Formally, the decision problem QBA-VMC (resp., QBA-VGS) takes as input a system K (resp., game G), a QBA (A, f), the oblivion bound B, and a value $a \in \mathbb{N} \cup \{\infty\}$,

and returns "Yes" if $val^{\max}_{\text{vbound}(A,f)}(K) \geq a$ (resp., $val^{\text{maxmin}}_{\text{vbound}(A,f)}(G) \geq a$). The decision problems QBA-IMC and QBA-IGS are defined analogously using $val^{\max}_{\text{ibound}(A,f)}(K)$ and $val^{\text{maxmin}}_{\text{ibound}(A,f)}(G)$. We give the oblivion bound as an input to the problems, because the value of $f(G)$ can be unboundedly larger than the descriptions of f and G. We assume that updates take unit time.

Theorem 5.1 *(1) QBA-VMC is PSPACE-complete and QBA-IMC is EXPTIME-complete. (2) QBA-VGS and QBA-IGS are EXPTIME-complete. (3) Let $|G|$ be the size of the structure and $|Q|$ the automaton size for (A, f) and G. Let $S = |Q| \cdot |G| \cdot (f(G) + 2)^k)$. QBA-VMC and QBA-VGS can be solved in time $O(S)$ and $O(S^2)$ respectively. QBA-IMC and QBA-IGS can be solved in time $O(|Q| \cdot |G| \cdot f(G))$.*

Note that these complexity results reflect the sizes of the state space in which the solution lies. In practice, however, the reachable state space can be much smaller. Hence, on-the-fly state space exploration can be used instead of constructing the entire state space *a priori*. The following examples show that our approach, while being generic and capturing several interesting quantitative verification problems [42] as special cases, still remains amenable to efficient analysis.

Example 5.7 (Fair maximum) Consider the deterministic QBA (A, f) with value-bound function f from Example 5.5, which computes the property *fm* from Example 5.2. This property is exactly the winning condition for the "threshold Büchi games" described in [42]. For a game G, the state space with the value bound has size $O(|G| \cdot |Q| \cdot \Delta)$, where Δ is the maximal value of proposition p in G. This is exponential in $|G|$. However, the iteration bound for this problem is $|G|$, and this gives an $O(|G|^2)$ algorithm, which is the same complexity as the algorithm of [42].[3] ∎

[3]However, computing an iteration-bound function automatically using the optimal value-bound function would lead to a suboptimal iteration-bound function $g(G) = |G| \cdot |Q| \cdot \Delta$.

Example 5.8 (Peak running total) The property *prt* from Example 5.4 is exactly the winning condition for the "energy games" of [42]. This property can be computed by a deterministic QBA with two registers and value-bound function $f(G) = |G| \cdot \Delta$, where Δ is the maximal value of p in G. A game-solving algorithm based on value bounds would require time $O(|G|^6 \cdot \Delta^4)$, whereas an algorithm designed specifically to solve this game [42] runs in time $O(|G|^3 \cdot \Delta)$. However, even for this problem, our generic approach, using the optimal iteration-bound function $h(G) = |G|^2 \cdot \Delta$ achieves the best known complexity of $O(|G|^3 \cdot \Delta)$. ∎

In the special case of monotonic automata, efficient iteration bounds can be automatically derived from value bounds. Consider a structure G with N states and a monotonic QBA (A, f) with value-bound function f, location set Q, and k registers. Since the value of each register only increases, within $|Q| \cdot k \cdot N \cdot f(G)$ steps of every run of A over a trace generated by G, either an automaton configuration repeats, or there is a register such that the value of the register has crossed the threshold $f(G)$. Thus $val^{\max}_{\mathrm{vbound}(A,f)}(G)$ is achieved by a run within $|Q| \cdot k \cdot N \cdot f(G)$ steps. Since we only require the monotonicity of the registers in the limit, this observation can be generalized to *reversal-bounded automata* [136], where a bounded number of switches between increasing and decreasing modes of the registers are allowed.

Proposition 5.1 *Let A be a monotonic automaton with location set Q and k registers, let $f\colon \mathcal{G} \to \mathbb{N}$ be a recursive function, and let $g(G) = |Q| \cdot k \cdot N \cdot f(G)$ for all structures G with N states. Then $val^m_{\mathrm{vbound}(A,f)}(G) = val^m_{\mathrm{ibound}(A,g)}(G)$ for all structures G and $m \in \{\max, \min, \mathrm{maxmin}\}$.*

As with the other components of a quantitative automaton, the designer has to provide the bound function f. Unfortunately, the task of providing a good value or

iteration bound function f, that is, an f that satisfies $val^m_A(G) = val^m_{\mathrm{bound}(A,f)}(G)$ for all structures G, cannot be automated.

Proposition 5.2 *There is a class of update functions involving only increment operations and equality testing on registers, such that the following two problems are undecidable: (1) given an automaton A, determine if there is a recursive function f such that $val^{\max}_A(K) = val^{\max}_{\mathrm{vbound}(A,f)}(K)$ for all systems K; (2) given a QBA (A, f), determine if $val^{\max}_A(K) = val^{\max}_{\mathrm{vbound}(A,f)}(K)$ for all systems K.*

5.4 The Quantitative-Bound μ-Calculus

We now provide an alternative formalism for defining quantitative properties: a fixpoint calculus. Our integer-based μ-calculus generalizes the classical μ-calculus [103], and provides an alternative set of iterative algorithms for model checking and game solving.

Unbounded formulas. Let P be a set of propositions, let $\mathcal{X}$ be a set of variables, and let $\mathcal{F}$ be a set of recursive functions from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$. We require that $\max, \min \in \mathcal{F}$. The formulas of the *quantitative μ-calculus*[4] are defined as

$$\varphi ::= k \mid p \mid X \mid upd(\varphi, \varphi) \mid pre(\varphi) \mid \mu[(X, \varphi), \ldots, (X, \varphi)] \mid \nu[(X, \varphi), \ldots, (X, \varphi)],$$

where k ranges over the constants in $\mathbb{N} \cup \{\infty\}$, p over the propositions in P, X over the variables in $\mathcal{X}$, and *upd* over the functions in $\mathcal{F}$. If *pre* ranges over the set $\{Epre, Apre\}$ of existential and universal next-time operators, we obtain the *system calculus*; if *pre* ranges over the set $\{Cpre_1, Cpre_2\}$ of player-1 and

[4] This is different from the μ-calculi over probabilistic systems defined by [97, 58, 113].

player-2 controllable next-time operators, we obtain the *game calculus*. Each least-fixpoint subformula $\mu[(X_1, \varphi_1), \ldots, (X_m, \varphi_m)]$ and each greatest-fixpoint subformula $\nu[(X_1, \varphi_1), \ldots, (X_m, \varphi_m)]$ binds a set $\{X_1, \ldots, X_m\}$ of variables. A formula φ is *closed* if all occurrences of variables in φ are bound.

The formulas of the quantitative μ-calculus are interpreted over quantitative structures (systems or games). Consider a game $G = (S, S_1, S_2, \delta, s_0, \langle\cdot\rangle)$. A *quantitative valuation* (*valuation*, for short) is a function θ: $S \to \mathbb{N} \cup \{\infty\}$ that maps each state s to a natural number or infinity. We write Θ for the set of valuations. The semantics $[\![\varphi]\!]$ of a closed formula φ over the structure G is a valuation in Θ, which is defined as follows. An *environment* $\mathbb{E}$: $\mathcal{X} \to \Theta$ maps each variable to a valuation. Given an environment $\mathbb{E}$, we write $\mathbb{E}[X := \theta]$ for the environment that maps X to θ, and maps each $Y \in \mathcal{X} \setminus \{X\}$ to $\mathbb{E}(Y)$. Each update function $upd \in \mathcal{F}$ defines a transformer $[upd]$: $\Theta \times \Theta \to \Theta$ that maps a pair of valuations to the valuation obtained by the point-wise application of upd. Each next-time operator pre defines a transformer $[pre]$: $\Theta \to \Theta$ that maps valuations to valuations. Specifically, $[Epre](\theta)(s) = \max\{\theta(s') \mid (s, s') \in \delta\}$; $[Apre](\theta)(s) = \min\{\theta(s') \mid (s, s') \in \delta\}$; $[Cpre_1](\theta)(s) = [Epre](\theta)(s)$ if $s \in S_1$, and $[Cpre_1](\theta)(s) = [Apre](\theta)(s)$ if $s \in S_2$; $[Cpre_2](\theta)(s) = [Apre](\theta)(s)$ if $s \in S_1$, and $[Cpre_2](\theta)(s) = [Epre](\theta)(s)$ if $s \in S_2$. For an environment $\mathbb{E}$, the semantics $[\![\varphi]\!]_{\mathbb{E}}$ of a (not necessarily closed) formula φ over G is defined inductively:

$$
\begin{aligned}
&[\![k]\!]_{\mathbb{E}}(s) \;=\; k; \quad [\![p]\!]_{\mathbb{E}}(s) \;=\; \langle s\rangle(p); \quad [\![X]\!]_{\mathbb{E}}(s) \;=\; \mathbb{E}(X)(s);\\
&[\![upd(\varphi_1, \varphi_2)]\!]_{\mathbb{E}}(s) \;=\; [upd]([\![\varphi_1]\!]_{\mathbb{E}}, [\![\varphi_2]\!]_{\mathbb{E}})(s);\\
&[\![pre(\varphi)]\!]_{\mathbb{E}}(s) \;=\; [pre]([\![\varphi]\!]_{\mathbb{E}})(s);\\
&[\![\mu[(X_1, \varphi_1), \ldots, (X_m, \varphi_m)]]\!]_{\mathbb{E}}(s) \;=\; \limsup\{\mathbb{E}^{\mu}_j(X_1)(s) \mid j \geq 0\};\\
&[\![\nu[(X_1, \varphi_1), \ldots, (X_m, \varphi_m)]]\!]_{\mathbb{E}}(s) \;=\; \limsup\{\mathbb{E}^{\nu}_j(X_1)(s) \mid j \geq 0\}.
\end{aligned}
$$

The environment $\mathbb{E}_j^\mu$ is defined inductively by $\mathbb{E}_0^\mu(X_i) = (\lambda s.\, 0)$ and $\mathbb{E}_{j+1}^\mu(X_i) = [\![\varphi_i]\!]_{\mathbb{E}_j^\mu}$ for all $1 \leq i \leq m$; and $\mathbb{E}_j^\mu(Y) = \mathbb{E}(Y)$ for all $Y \in \mathcal{X} \setminus \{X_1, \ldots, X_m\}$ and $j \geq 0$. The environment $\mathbb{E}_j^\nu$ is defined like $\mathbb{E}_j^\mu$ except that $\mathbb{E}_0^\nu(X_i) = (\lambda s.\, \infty)$ for all $1 \leq i \leq m$. For monotone boolean formulas, the limsup semantics coincides with the usual fixpoint semantics of the μ-calculus [103]. For a closed formula φ, we define $[\![\varphi]\!]$ as $[\![\varphi]\!]_{\mathbb{E}}$, for an arbitrary environment $\mathbb{E}$. Given a structure G, the closed formula φ *specifies* the valuation $[\![\varphi]\!](G) = [\![\varphi]\!](s_0)$, where s_0 is the initial state of G.

Bound functions. A *quantitative-bound μ-formula* (QBF) (φ, f) consists of a quantitative μ-formula φ and a recursive function $f\colon \mathcal{G} \to \mathbb{N}$ that provides a bound $f(G)$ on the number of iterations necessary for evaluating μ and ν subformulas on any given structure G. The semantics $[\![(\varphi, f)]\!]_{\mathbb{E}}$ of a QBF (φ, f) over a structure G is defined like the semantics of the unbounded formula φ except that each fixpoint subformula is computed by unrolling the fixpoint only $O(f(G))$ times. Formally, a variable X is *$f(G)$-stable* at a state s with respect to a sequence $\{\mathbb{E}_j \mid j \geq 0\}$ of environments if $\max\{\mathbb{E}_j(X)(s) \mid f(G) \leq j \leq 2 \cdot f(G)\} = \max\{\mathbb{E}_j(X)(s) \mid 2 \cdot f(G) \leq j \leq 3 \cdot f(G)\}$. We define $[\![\mu[(X_1, \varphi_1), \ldots, (X_m, \varphi_m)], f]\!](s)$ to be $\max\{\mathbb{E}_j^\mu(X_1)(s) \mid f(G) \leq j \leq 2 \cdot f(G)\}$ if all variables X_i, for $1 \leq i \leq m$, are $f(G)$-stable with respect to $\{\mathbb{E}_j^\mu \mid j \geq 0\}$; otherwise $[\![\mu[(X_1, \varphi_1), \ldots, (X_m, \varphi_m)], f]\!](s) = \infty$. The semantics $[\![\nu[(X_1, \varphi_1), \ldots, (X_m, \varphi_m)], f]\!]$ of greatest-fixpoint subformulas is defined analogously, using the sequence $\{\mathbb{E}_j^\nu \mid j \geq 0\}$ of environments instead. A QBF formula (φ, f) defines an iterative algorithm for computing the valuation $[\![(\varphi, f)]\!](G)$ for any given structure G. Assuming updates take unit time, we can compute $[\![(\varphi, f)]\!](G)$ in $O(f(G)^\ell)$ time, where ℓ is the alternation depth of φ (i.e., the maximal number of alternations between occurrences of μ and ν operators; for a precise definition see [65]).

We now give examples for which a QBF (φ, f) can be found to specify the same

property as the unbounded formula φ over all structures; that is, $[\![(\varphi, f)]\!](G) = [\![\varphi]\!](G)$ for all structures G. We use addition, subtraction, and comparison as update functions in $\mathcal{F}$, and we use the natural numbers 0 and 1 to encode booleans. For instance, we write $\varphi_1 = \varphi_2$ for $\min(\varphi_1 \leq \varphi_2, \varphi_2 \leq \varphi_1)$, and $\neg\varphi_1$ for $1 - \varphi$. The case formula $\mathrm{case}\{(\psi_1, \varphi_1), \ldots, (\psi_n, \varphi_n)\}$ stands for $\max(\min(\psi_1, \varphi_1), \ldots, \min(\psi_n, \varphi_n))$, where the n-ary max operator is obtained by repeated application of the binary max operator. In order to relate the branching-time framework of the quantitative μ-calculus to the linear-time framework of quantitative properties (and quantitative automata), we say that the closed QBF (φ, f) *computes* the property π if for all structures G, $[\![(\varphi, f)]\!](G) = \sup\{\pi(w) \mid w \text{ is a trace generated by } G\}$. In this way, linear and branching time are related *existentially* (through sup rather than inf); hence we use only the *Epre* operator to compute properties. Alternately, we could define a universal semantics where $[\![(\varphi', f)]\!](G) = \inf\{\pi(w) \mid w \text{ is a trace generated by } G\}$, and the *Apre* operator is used.

Example 5.9 (Fair maximum) Recall the property *fm* from Example 5.2. The property *fm* is computed over all structures G by the QBF (φ, f) with $\varphi = \mu[(X, \min\{\max\{p, X, \min\{\mathit{Epre}(X), Z\}\}, Z\})]$, where $Z = \nu[(X, \mu[(Y, \mathit{Epre}(\max\{\min\{q, X\}, Y\})])]$, and $f(G) = N$, where N is the number of states of G. Since the longest simple path in G has length at most $N - 1$, every fixpoint is found in N iterations or less. ∎

Example 5.10 (Lifetime) Over all structures G with N states, the property lt_a from Example 5.3 is computed by the QBF (φ, f) with $\varphi = \mu[(X, \mathrm{case}\{((c = 0) \wedge (p + \mathit{Epre}(Y) \leq a), X + 1), (c \neq 0, X + 1), (1, X)\}), (Y, \mathrm{case}\{(((c = 0) \wedge (p + \mathit{Epre}(Y) \leq a)), p + \mathit{Epre}(Y)), (c \neq 0, \mathit{Epre}(Y) - a), (1, Y)\})]$ and $f(G) = N \cdot a + N \cdot (N + 1) \cdot \Delta$, where Δ is the maximal value of the proposition p in G. If a fixpoint is not reached in

$N \cdot a + N \cdot (N+1) \cdot \Delta$ iterations, then there is a reachable cycle Γ in G with nonpositive resource consumption, and repeated traversal of Γ ensures an infinite lifetime. ∎

Example 5.11 (Peak running total) Over all structures G with N states and maximal value Δ for the proposition p, the property *prt* from Example 5.4 is computed by the QBF (φ, f) with $\varphi = (\mu[(X, \text{case}\{(c = 0, p + \max\{0, \mathit{Epre}(X)\}), (c \neq 0, \max\{0, \mathit{Epre}(X)\} - p)\})], f)$ and $f(G) = N \cdot \Delta$. If a fixpoint is not reached in $N \cdot \Delta$ iterations, then there is no reachable cycle with nonpositive resource consumption, and it is not possible to traverse G forever starting with a finite amount of resources. ∎

From automata bounds to μ-calculus bounds. We establish the connection between properties specified by quantitative automata (a linear-time formalism) and those computed by the quantitative μ-calculus (a branching-time formalism). We show that every deterministic QBA can be converted to a QBF that computes the same property over all systems. This provides an alternative algorithm for quantitative model checking. We then show that the construction is robust [56], and hence, the resulting QBF can also be used for game solving. To formalize this, we define a quantitative μ-calculus over traces, extending the boolean linear-time μ-calculus [133]. The *quantitative-bound trace formulas* (QBTs) are identical to the quantitative-bound μ-formulas, except that they contain the single next-time operator *Pre*. A QBT is interpreted over the traces w generated by a given structure G. To define $[\![(\varphi, f)]\!](w)$ formally, we view the trace $w = o_0 o_1 o_2 \ldots$ as an infinite-state system without branching, analogous to the boolean definition in [56]. However, even though w is infinite-state, the evaluation of every fixpoint subformula in φ is bounded by $f(G)$, which is finite.

Consider a structure K, a game G, and a QBT (φ, f). The system value $val^{\max}_{(\varphi,f)}(K)$ (resp., $val^{\min}_{(\varphi,f)}(K)$) is the supremal (resp., infimal) value of the formula (φ, f) over all traces generated by K. Formally, $val^{\max}_{(\varphi,f)}(K) = \sup\{[\![(\varphi, f)]\!](w) \mid w \text{ is a trace generated by } K\}$, and $val^{\min}_{(\varphi,f)}(K)$ is the inf of the same set. For strategies $\xi_1 \in \Xi_1$ and $\xi_2 \in \Xi_2$, define $val_{(\varphi,f)}(\xi_1, \xi_2) = [\![(\varphi, f)]\!](\langle t_{\xi_1,\xi_2}\rangle)$. The game value $val^{\max\min}_{(\varphi,f)}(G) = \sup_{\xi_1\in\Xi_1} \inf_{\xi_2\in\Xi_2} val_{(\varphi,f)}(\xi_1, \xi_2)$ is the supremal value that player 1 can achieve against all player-2 strategies. The following two theorems generalize the results of [56] from boolean to quantitative verification: Theorem 5.2 establishes the connection between deterministic QBAs and QBTs; Theorem 5.3 presents a necessary and sufficient criterion, called *robustness*, when a QBT can be used for game solving. Moreover, the QBT constructed in Theorem 5.2 is robust. Given a QBT (φ, f), let $(\varphi[Epre], f)$ (resp., $(\varphi[Apre], f)$) be the QBF that results by replacing all occurrences of the next-time operator *Pre* with *Epre* (resp., *Apre*).

Theorem 5.2 *Every deterministic QBA (A, f) can be translated into a QBT (φ, g) such that for all systems K, both $val^{\max}_{(A,f)}(K) = val^{\max}_{(\varphi,g)}(K) = [\![(\varphi[Epre], g)]\!](K)$ and $val^{\min}_{(A,f)}(K) = val^{\min}_{(\varphi,g)}(K) = [\![(\varphi[Apre], g)]\!](K)$.*

Theorem 5.3 *Given a QBT (φ, f), the following two conditions, called* robustness*, are equivalent. (1) For all systems K, both $val^{\max}_{(\varphi,f)}(K) = [\![(\varphi[Epre], f)]\!](K)$ and $val^{\min}_{(\varphi,f)}(K) = [\![(\varphi[Apre], f)]\!](K)$. (2) For all games G, $val^{\max\min}_{(\varphi,f)}(G) = [\![(\varphi[Cpre_1], f)]\!](G)$.*

Theorem 5.2 is proved using a standard (boolean) construction of a fixpoint formula from an automaton [51]. Theorem 5.3 follows from the existence of finite-memory optimal strategies for QBTs.

5.5 Unbounded Quantitative Automata and their Expressiveness

By a simple counting argument, there are properties that cannot be computed by quantitative automata. We now give some finer classification of the expressiveness of various subclasses. We denote the class of automata with update functions restricted to only linear combinations of the registers and input as *affine quantitative automata*. All our examples fall into the class of affine quantitative automata. The bound k on the number of registers naturally induces an expressiveness hierarchy in the class of affine quantitative automata. Let Π_k^D (respectively, Π_k^N) denote the set of properties that can be computed by deterministic (respectively, nondeterministic) affine quantitative automata with k registers. Also there are properties that can be expressed by deterministic affine quantitative automata but cannot be expressed by quantitative-bound automata.

Theorem 5.4 (Expressiveness hierarchy) *(1)* $\Pi_k^D \subsetneq \Pi_{k+1}^D$ *and* $\Pi_k^N \subsetneq \Pi_{k+1}^N$, *(2)* $\Pi_k^D \subsetneq \Pi_k^N$, *and (3)* $\Pi_2^N \not\subseteq \bigcup_{k\in\mathbb{N}} \Pi_k^D$. *(4) There exists a property* $\mathcal{P} \in \bigcup_k \Pi_k^D$ *such that there is no quantitative-bound automaton* (A, f) *with* $\Pi(A, f) = \mathcal{P}$.

Example 5.12 shows that infinite-memory strategies may be required to achieve the value for games with quantitative objectives. This fact is in contrast to boolean properties, where games with objectives given by ω-automata admit finite-memory determinacy.

Example 5.12 (Maximum over non-regular traces) We show that it is not necessary that the value of a system is achieved over traces of the form $u \cdot v^\omega$. This is in

contrast with Büchi automata on infinite traces where if the language accepted by a Büchi automaton is not empty, then it accepts a trace of the form $u \cdot v^\omega$. Consider the automaton $A = \langle \{q_0, q_1, q_{halt}\}, 2, q_0, \delta \rangle$, where for all $x \in \mathcal{O}$, we have:

- $\delta(q_0, x) = \{(q_1 \wedge x \neq 1 \wedge R'[0] := 1, R'[1] := 0), (q_{halt} \wedge x = 1 \wedge R'[0] := 0 \wedge R'[1] := 0)\}$.
- $\delta(q_1, x) = \{(q_1, x = 1 \wedge R'[1] := R[1] + 1), (q_1, x \neq 1 \wedge R[1] = R[0] \wedge R'[0] := R[0] + 1 \wedge R'[1] := 0), (q_{halt}, x \neq 1 \wedge R'[1] \neq R[0] \wedge R'[1] := 0)\}$.

The automaton checks whether the next sequence of successive 1's in the input is of length $R[0]$. If yes, it increments $R[0]$ by 1 and continues. Otherwise, it resets the register $R[1]$ to 0 and halts. The property f that the automaton specifies can be described as follows: given a trace $w \in (0+1)^\omega$ we have $f(w) = \max\{i \mid w \in 0^+\ 1\ 0^+\ 1\ 1\ 0^+1\ 1\ 1\ 0 \ldots 0^+1^i\ 0\ (0+1)^\omega\}$. It is easy to see that the maximum will be achieved over traces of the form $0^+\ 1\ 0^+\ 1\ 1\ 0^+\ 1\ 1\ 1\ 0^+\ \ldots$ which is not regular. ∎

Theorem 5.5 (Nonexistence of finite-memory strategies) *Given an affine quantitative automaton A and a game G, the strategy $\hat{\xi}$ such that $\inf_{\xi_2 \in \Xi_2} val_A(\hat{\xi}, \xi_2) = val_{\text{maxmin}}(A, G)$ may require infinite memory.*

5.6 Conclusion

We generalized the boolean verification framework of transition systems, traces, languages, automata, games, and μ-calculus to a quantitative, integer-based setting. Our framework allows the natural expression of properties about quantitative resources such as power and buffer size. The integer-based verification problems reduce

to dynamic programming, that is, the iterative evaluation of a set of integer constraints over a finite state space. While these problems are in general undecidable, we showed that for many properties of practical interest, a bound function can be specified, which guarantees the finite convergence of the iterative computation. The novelty of bound functions is that they assign different bounds to different systems, rather than a fixed bound for a given property. We showed that the resulting generic algorithms match the best-known property-specific algorithms. In other words, we shifted the burden in the verification of a quantitative property from the task of designing an algorithm and proving its termination to the often simpler task of providing a quantitative automaton together with a value or iteration bound. In the final two sections, section, we showed that many but not all properties of the boolean verification framework, e.g. the close correspondence between model-checking and game-solving, carry over to the quantitative case. For example, the close correspondence between model-checking and game-solving carries over for quantitative-bound properties, but not for unbounded quantitative properties, because in the latter case optimal strategies may require infinite memory.

Acknowledgements

The work reported in this chapter was conducted jointly with Krishnendu Chatterjee, Prof Thomas A. Henzinger, Prof Orna Kupferman, and Prof Rupak Majumdar. This research was funded by Prof Henzinger supported in part by the ONR grant N00014-02-1-0671, the AFOSR MURI grant F49620-00-1-0327, the NSF ITR CHESS grant CCR-0225610, and the NSF grants CCR-0234690, and CCR-0427202. This

chapter is based on a paper [39] presented at CHARME 2005, published by Springer; copyright held by IFIP International Federation for Information Processing[5], 2005.

[5] http://www.springerlink.com

Chapter 6

Function Interfaces and Software Partitioning

A key problem for effective unit testing is the difficulty of partitioning large software systems into appropriate units that can be tested in isolation. We present an approach that identifies control and data inter-dependencies between software components using static program analysis, and divides the source code into units where highly-intertwined components are grouped together. Those units can then be tested in isolation using automated test generation techniques and tools, such as dynamic software model checkers. We discuss preliminary experimental results showing that automatic software partitioning can significantly increase test coverage without generating too many false alarms caused by unrealistic inputs being injected at interfaces between units.

6.1 Introduction

Today, testing is the primary way to check the correctness of software. Correctness is even more important to determine in the case of embedded software, where reliability and security are often key features. Billions of dollars are spent every year on testing in the software industry as a whole as testing usually accounts for about 50% of the cost of software development [119]. Yet software failures are numerous and their cost to the world's economy can also be counted in billions of dollars [125].

An approach (among others) that has potential to significantly improve on the current state of the art consists of automatically generating test inputs from a static or dynamic program analysis in order to force program executions towards specific code statements. *Automated test generation* from program analysis is an old idea (e.g., [101, 119, 102, 62]), which arguably has had a limited practical impact so far, perhaps due to the lack of usable, industrial-strength tools implementing this approach. Recently, there has been a renewed interest for automated test generation (e.g., [32, 28, 134, 49, 83, 37]). This renewed interest can perhaps be attributed to recent progress on software model checking, efficient theorem proving, static analysis and symbolic execution technology, as well as recent successes in engineering more practically-usable static-analysis tools (e.g., [36, 86]) and the increasing computational power available on modern computers.

A new idea in this area is Directed Automated Random Testing (DART) [83], which fully automates software testing by combining three main techniques: (1) *automated* extraction of the interface of a program with its external environment using static source-code parsing; (2) automatic generation of a test driver for this interface that performs *random* testing to simulate the most general environment the program

can operate in; and (3) dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs that *direct* the execution along alternative program paths. DART can detect standard errors such as program crashes, assertion violations, and non-termination, and can also be used in conjunction with complementary run-time checking tools for detecting memory allocation problems (e.g., [88, 121, 105, 4]). During testing (Step 3), DART performs a *directed search*, a variant of dynamic test generation (e.g., [102, 85]). Starting with a random input, a DART-instrumented program calculates during each execution an input vector for the next execution. This vector contains values that are the solution of *symbolic constraints* gathered from predicates in branch statements during the previous execution. The new input vector attempts to force the execution of the program through a new path. By repeating this process, a directed search attempts to force the program to sweep through all its feasible execution paths, in a style similar to *systematic testing* and *dynamic software model checking* [81].

When applied to large programs, a directed search is typically incomplete due to combinatorial explosion and because of the presence of complex program statements or external function calls for which no symbolic constraint can be generated. It is worth emphasizing that these two limitations – scalability and limited symbolic-reasoning capabilities – are inherent to *all* automated test-generation techniques and tools, whether static (e.g., [32, 28, 134, 49]) or dynamic (e.g., [83, 37]). This explains why *automated test generation typically cannot achieve 100% code coverage* in practice, independently of how code coverage is defined (e.g., coverage of all program statements, branches, paths, etc.).

An idea to alleviate these limitations is to partition a large program into several smaller units that are then tested in isolation to search for program bugs such as

crashes and assertion violations. As an extreme, every function in a program could be tested one-by-one in isolation using automated test generation. While this testing strategy can dramatically increase code coverage, automated test generation will also typically generate lots of unrealistic inputs which in turn will trigger many unrealistic behaviors and spurious bugs, i.e., *false alarms*. This is because function testing ignores all the dependencies (preconditions implied by the possible calling contexts) between functions.

In this chapter, we investigate *how to partition a program* into a set of units such that testing those units in isolation *maximizes overall code coverage while minimizing the number of false alarms*. We propose a two-step heuristic approach to this problem. First, we perform a light-weight static source-code analysis to identify the *interface* of every function in a program. Second, we divide the program into units where *highly-intertwined functions are grouped together*, hence hiding complex interfaces inside units. We have developed a prototype tool implementing these ideas for the C programming language. We present results of experiments performed on oSIP, an open-source implementation of the Session Initiation Protocol *embedded* in many IP phones. Those experiments show that automatic software partitioning using the above simple strategy can significantly increase test coverage without generating too many false alarms.

The rest of this chapter is organized as follows. We start in Section 6.2 with some background definitions and assumptions, and define the software partitioning problem in this context. The two steps of our approach to the partitioning problem are then discussed in Sections 6.3 and 6.4, on interfaces and partitioning algorithms respectively. The partitioning algorithms of Section 6.4 have been implemented in a prototype tool for partitioning C programs, and results of experiments are presented

in Section 6.5. Other related work is discussed in Section 6.6 and we conclude in Section 6.7.

6.2 The Software Partitioning Problem

A program P is defined as a set of functions $\mathcal{F}$ associated with a given function $main \in \mathcal{F}$ which is always executed first when running the program. A *call graph* $\mathcal{G}$ over a set of *functions* $\mathcal{F}$ is a graph $\mathcal{G} = (V, E)$ with a set of *vertices* $V = \mathcal{F}$ and a set of *edges* $E \subseteq V \times V$ such that $(f, g) \in E$ if there is a call to function g in function f. By transitive closure, we say that a function f *calls* a function g in a call graph $\mathcal{G}$ if there is a sequence $\sigma = s_0, s_1, \ldots, s_k$ such that $s_0 = f$, and $s_k = g$, and $(s_i, s_{i+1}) \in E$ for $i \in \{0, 1, \ldots, k-1\}$; the *path* from f to g is the sequence σ, and its *length* is k. A function f *directly calls* a function g in $\mathcal{G}$ if there is a path from f to g of length 1 in $\mathcal{G}$.

A path $\sigma = s_0, s_1, \ldots, s_k$ from f to g *lies in* a set S if $s_i \in S$ for all $i \in \{0, 1, \ldots, k\}$. A set of functions S is *convex* with respect to a function $f \in S$ if for every function g in S called by f, there is a path from f to g that lies in S. Given a set S of functions, a function $f \in S$ is called an *entry point* of S if every function $g \in S$ is called by f, and S is convex with respect to f. A *unit* $u = (S, f)$ is a set of functions S with an entry point f.

Given a call graph $\mathcal{G}$ over a set of functions $\mathcal{F}$, a *partition* $\mathcal{P}$ of $\mathcal{G}$ is a set of units $\mathcal{P} = \{(S_0, f_0), (S_1, f_1), \ldots, (S_k, f_k)\}$ such that $\bigcup\{S_i \mid 0 \leq i \leq k\} = \mathcal{F}$, and $S_i \cap S_j = \emptyset$ for all $0 \leq i, j \leq k$ such that $i \neq j$. We assume that the pair ($\mathcal{F}$, *main*) is a unit, i.e., *main* is an entry point of $\mathcal{F}$. Trivially, for any function f, $(\{f\}, f)$ is always a unit.

In order to test program P, automated test generation techniques and tools can

be applied to generate inputs at the program interface in order to drive its execution along as many program statements/branches/paths as possible. However, testing large programs is difficult because of the limited reasoning capabilities of automated test generation techniques and expensive because of the prohibitive number of statements, branches and paths in large programs. Consequently, viewing the entire program P as a single unit usually leads to poor code coverage (as will be shown experimentally later) if P is large and non-trivial. Let us call this form of testing *monolithic testing*.

On the other hand, automated test generation can also be applied to all the individual functions $f \in \mathcal{F}$ of P, one by one. This strategy can dramatically increase code coverage but automated test generation will typically generate lots of unrealistic inputs which in turn will trigger many unrealistic behaviors and hence many false alarms. Indeed, such a *piecemeal testing* ignores all the dependencies (preconditions implied by the possible calling contexts) between functions. For instance, a function f taking as input argument a pointer to a data structure may assume that this pointer is always non-null. While this assumption may be true whenever f is called by some function in $\mathcal{F}$, piecemeal testing may very well provide a null value for this argument when calling f, which may lead to a crash if the pointer is dereferenced inside f. We call such a spurious error a *false alarm*.

We define the *software partitioning problem* as follows:

> how to partition a given program P satisfying the above assumptions into a set of units such that testing those units in isolation maximizes code coverage while minimizing the number of false alarms?

Our definition is intentionally general to accommodate various specific ways to measure code coverage or identify/define false alarms.

In this chapter, we present a two-step heuristic approach to the software partitioning problem. First, we propose to perform a light-weight static source-code analysis to identify the *interface* of every function in a program P, which enables the definition and measurement of the *complexity* of those interfaces. Second, we discuss several clustering algorithms which group together functions whose joint interface is complex, hence hiding that interface inside the resulting unit. The result is a partition of the program P where highly intertwined functions are grouped together in units.

The two steps of our approach are discussed in the next two sections.

6.3 Interfaces

In principle, the interface of a function describes all possible avenues of exchange of information between the function and its environment: arguments, return values, shared variables, and calls to other functions. The interface of a unit is defined as the interface of the composition of the functions in that unit.

In practice, the precise interface of functions described in full-fledged programming languages like C or C++ can be hard to determine statically due to unknown control flow (e.g., in the presence of function pointers), unbounded data structures, side-effects through global variables, etc. We therefore use approximate interface representations.

Control interfaces track control dependencies across function and unit boundaries. Given a program P defined by a set $\mathcal{F}$ of functions, the *boolean control interface* $\mathcal{C}$ of a unit $u = (S, f_S)$ of P is a tuple $(S, \mathcal{O}, \mathcal{I})$ where $\mathcal{O} : S \times (\mathcal{F} \setminus S)$ is a relation mapping every function $f \in S$ to the functions $g \in \mathcal{F} \setminus S$ that are directly called by

f, and $\mathcal{I} : (\mathcal{F} \setminus S) \times S$ is a relation mapping every function $g \in \mathcal{F} \setminus S$ to the functions $f \in S$ directly called by g.

By extension, given a program P defined by a set $\mathcal{F}$ of functions, the *weighted control interface* $\mathcal{C}_w$ of a unit $u = (S, f_S)$ of P is a tuple $(S, \mathcal{O}_w, \mathcal{I}_w)$ where $\mathcal{O}_w : S \times ((\mathcal{F} \setminus S) \times \mathbb{N})$ is a relation mapping every function $f \in S$ to the pairs (g, n) where $g \in \mathcal{F} \setminus S$ is a function directly called by f, and n is the number of calls to g in the code describing f, and $\mathcal{I}_w : (\mathcal{F} \setminus S) \times (S \times \mathbb{N})$ is a relation mapping every function $g \in \mathcal{F} \setminus S$ to the pairs (f, n) where $f \in S$ is a function directly called by g and n is the number of calls to f in the code describing g.

Two boolean (weighted) control interfaces $\mathcal{C}_1 = (S_1, \mathcal{O}_1, \mathcal{I}_1)$ and $\mathcal{C}_2 = (S_2, \mathcal{O}_2, \mathcal{I}_2)$ are *compatible* (denoted $comp(\mathcal{C}_1, \mathcal{C}_2)$) if $S_1 \cap S_2 = \emptyset$. Given two compatible boolean control interfaces $\mathcal{C}_1 = (S_1, \mathcal{O}_1, \mathcal{I}_1)$ and $\mathcal{C}_2 = (S_2, \mathcal{O}_2, \mathcal{I}_2)$, their *composition* (denoted $\mathcal{C}_1 \parallel \mathcal{C}_2$) is the boolean control interface $\mathcal{C}_c = (S_c, \mathcal{O}_c, \mathcal{I}_c)$ where $S_c = S_1 \cup S_2$, $\mathcal{O}_c = \{(f, g) \in \mathcal{O}_1 \cup \mathcal{O}_2 | g \in \mathcal{F} \setminus (S_1 \cup S_2)\}$ and $\mathcal{I}_c = \{(f, g) \in \mathcal{I}_1 \cup \mathcal{I}_2 | f \in \mathcal{F} \setminus (S_1 \cup S_2)\}$ (calls from S_1 and S_2 to each other are hidden by the composition). Similarly, the composition of two compatible weighted control interfaces is defined as the weighted control interface $\mathcal{C}_c = (S_c, \mathcal{O}_c, \mathcal{I}_c)$ where $S_c = S_1 \cup S_2$, $\mathcal{O}_c = \{(f, (g, n)) \in \mathcal{O}_1 \cup \mathcal{O}_2 | g \in \mathcal{F} \setminus (S_1 \cup S_2)\}$ and $\mathcal{I}_c = \{(f, (g, n)) \in \mathcal{I}_1 \cup \mathcal{I}_2 | f \in \mathcal{F} \setminus (S_1 \cup S_2)\}$.

Richer interface representations can be defined by also taking into account other features of a function that are externally visible, such as the number of input arguments, the type of input arguments and return values for incoming or outgoing function calls, the sequencing of external calls made by a function (flow-sensitive interface representation), etc.

For simplicity, we consider in this work partitioning algorithms that make use of

information exposed by control interfaces only. Whether richer interface representations can lead to significantly better software partitioning is left for future work.

6.4 Software Partitioning Algorithms

In this section, we present partitioning algorithms that exploit the definitions of interfaces of the previous section in order to group together functions (atomic components) that share interfaces of complexity higher than a given threshold. Two notions of interface complexity are implicitly considered here: popularity and collaboration.

6.4.1 Callee Popularity

If a function f is called by many different caller functions, then it is likely that any specific calling context in which f is called is not particularly important. For instance, functions used to store or access values in data structures like lists or hash tables are likely to be called by several different functions. In contrast, functions performing sub-tasks specific to a given function may be called only once or only by that function. In the latter case, the caller and the callee are more likely to obey tacit rules when communicating with each other, and breaking the interface between them by placing these functions in different units may be risky. Thus, the more *popular* a function is, the more likely it is to have generic and loose relationships with each of its callers, and the less likely it is that they share overly intricate and detailed assumptions and guarantees about each other's behavior.

In what follows, let the *popularity* of a function f be the number of functions g that call f. (The popularity of a function could also be defined as the number of

Algorithm 3 PartitionCP(S)

Input: A set of control interfaces of a set S of functions

Output: A partition of S into a set U of units

Variables: *WeightedGraph* G

```
1:  G := PopularityGraph(S)
2:  while (¬ IsEmpty(G)) do
3:     c := ChooseCPCutoff(G)
4:     G' := FilterPopularEdges(G, c)
5:     while (¬ IsEmpty(G')) do
6:        t := TopLevel(G')
7:        u := Reachable(G', t)
8:        if (|u| > 1 or G' = G) then
9:           add u as a new unit in U
10:          G := RemoveNodes(G, u)
          end if
11:       G' := RemoveNodes(G', u)
       end while
    end while
```

syntactic calls to that function; we consider here the former, simpler definition in order to determine if it is already sufficient to obtain interesting results.)

Formalizing the above intuition, our first algorithm generates a partition of units in which functions f and g are more likely to be assigned to the same unit if f calls g and g is not very popular. Given the set of control interfaces of a set S of functions, the algorithm starts by invoking the *PopularityGraph* subroutine to compute a weighted directed graph $G = (V, E)$ with the set of nodes $V = S$, where the directed edges in E

denote calls between functions in S, and each edge weight is set to the popularity of the callee (i.e., the popularity of the function corresponding to the destination node of the edge).

Next, as long as the popularity graph $G = (V, E)$ is not empty, the algorithm proceeds with the following steps. Given a *cutoff policy*, it chooses a maximum popularity cutoff c by invoking the subroutine *ChooseCPCutoff*, and (temporarily) removes edges above that threshold by calling the subroutine *FilterPopularEdges*. The resulting subgraph G' is traversed top-down (or top-down in some spanning tree in case the subgraph is strongly connected). This traversal is performed by repeatedly invoking the subroutine *TopLevel* which returns a *top level node* in $G' = (V', E')$: a node v in V' is said to be a top level node if no other node u exists in V' such that $(u, v) \in E'$. Note that a (nonempty) directed graph has no top level node if and only if every node is part of some strongly connected component; in that case the subroutine *TopLevel* returns an arbitrary $v \in V'$. For each top level node t, the set of reachable nodes in G' from t is computed by invoking the subroutine *Reachable*. If this set of nodes is non-trivial (i.e., larger than one node), this set of nodes is defined as a unit. Nodes that would form trivial units (consisting of only themselves) are not allocated at this stage, but will be allocated at some subsequent iteration of the outer **while** loop with a possibly different (higher) popularity cutoff. Nodes corresponding to functions allocated to units are removed from G and from the subgraph G' using the *RemoveNodes* subroutine. When the inner **while** loop exits, a new iteration of the outer **while** loop may begin, and the entire process described above is repeated until all the functions have been allocated to some unit, at which point the outer **while** loop exits, and the algorithm terminates. At that point, every function in S has been allocated to exactly one unit in the resulting set U of units.

Algorithm 3 uses the following subroutines:

1. The subroutine *PopularityGraph* takes as input a set of (boolean) control interfaces of a set of functions S, and returns a weighted directed graph $G = (V, E)$. The graph is such that there is a vertex $v_f \in V$ for each function $f \in S$, and there is an edge $(v_f, v_g) \in E$ with weight w for each call from f to g in S, where w is the popularity of g.

2. The subroutine *IsEmpty* takes as input a graph $G = (V, E)$ and returns T if $V = \emptyset$, and F otherwise.

3. The subroutine *ChooseCPCutoff* takes as input a weighted directed graph G and returns a value c based on the *cutoff policy*. A cutoff policy is an external parameter to the algorithm. We considered and experimented with two types of cutoff policies in conjunction with this algorithm: policy `cp`n makes *ChooseCPCutoff* return the value n on the first invocation and the maximum weight in G on subsequent invocations, while policy `cpi` makes *ChooseCPCutoff* return the smallest weight in G.

4. The subroutine *FilterPopularEdges* takes as inputs a weighted directed graph $G = (V, E)$ and a value c. It returns a weighted directed graph $G' = (V, E')$ such that $E' \subseteq E$, and for all $e \in E$ of weight w, we have $e \in E'$ if and only if $w \leq c$.

5. The subroutine *TopLevel* takes as input a (nonempty) weighted directed graph $G' = (V', E')$ and returns any node v' such that there is no $v \in V'$ such that $(v, v') \in E'$. If such a node v' does not exist (i.e., every node in G' is part of some strongly connected component), then *TopLevel*(G') returns any $v' \in V'$.

6. The subroutine *Reachable* takes as inputs a graph $G = (V, E)$ and a node $t \in V$, and returns the set of nodes $V' \subseteq V$ reachable from t in G.

7. The subroutine *RemoveNodes* takes a graph $G = (V, E)$ and a set $u \subseteq V$ and returns a graph $G' = (V', E')$ such that $V' = V \setminus u$, and (V', E') is the subgraph of G induced by V'.

Theorem 6.1 *For a call graph $\mathcal{G}$ over a set of functions S and given a set of control interfaces of S, the algorithm* PartitionCP(S) *creates a partition of $\mathcal{G}$.*

Proof. We prove (i) that the algorithm terminates, and (ii) that when it has terminated, (a) every function in S is allocated to some unit u generated by the algorithm, and (b) that no function $f \in S$ is allocated to two units u and u' generated by it.

The subroutine *ChooseCPCutoff*, for all cutoff policies, eventually returns the maximum weight in G. Thus, the condition $G = G'$ in the **if** condition in the inner **while** loop is satisfied eventually. From that point onwards, at least one vertex is removed from the graphs G and G' in each iteration of the inner **while** loop. Since the graph G' is finite, the inner **while** loop must eventually exit when G' becomes empty. Also, since the graph G is finite, the outer **while** loop must eventually exit when G becomes empty. Thus, the algorithm terminates.

Since every function in S is a vertex in the popularity graph G, and since the outer loop runs until the graph G is empty (has no more vertices), and since a vertex is removed from G only if it is allocated to a generated unit, it follows that every function in S is allocated to some unit. For the part (ii)(b), we observe that, when a function f is allocated to a unit u (in line 9 of the algorithm), it is next removed from both G and G', and thus cannot be allocated to more than one unit.

6.4.2 Shared Code

If two functions f and g call many of the same functions, then it is likely that the higher-level operations they perform are functionally more related than with other functions that call a completely disjoint set of sub-functions. Therefore, functions that share a lot of sub-functions should perhaps be grouped in a same unit.

This intuition is formalized by our second partitioning algorithm, which generates a partition of units in which functions f and g are more likely to be assigned to the same unit if they have a relatively high degree of *collaboration*, where the degree of *collaboration* is the number of common functions called by both f and g.

Given a set of control interfaces of a set S of functions, the algorithm first creates a weighted undirected *collaboration graph* $W = (V, E)$ with $V = S$, and $E = (S \times S) \setminus \{(f, f) \mid f \in S\}$ (intuitively, there is an edge between any two distinct functions), and each edge weight is set to the number of sub-functions shared between the two functions the edge connect. Since an approach based on a single cutoff classifying inter-function collaboration into a boolean "high" or "low" is too coarse-grained, we instead propose a more general algorithm based on multiple collaboration thresholds. Given a *collaboration classification* policy (embodied by the *NumberOfCollaborationClasses* subroutine) denoting the number c of collaboration classes, the algorithm invokes the *CollaborationThresholds* subroutine to compute a set of c *collaboration thresholds* representing a sequence of *minimum collaboration levels* the algorithm will run through in descending order in subsequent iterations of the outer **while** loop, and edges representing collaboration levels below the minimum currently chosen will be (temporarily) removed by the *FilterLightEdges* subroutine invoked soon afterwards in the outer **while** loop. The outer **while** loop runs as long

as the current collaboration graph W is not empty. At the beginning of each iteration, the maximum value t in the current list L of collaboration thresholds is found, and passed to the *FilterLightEdges* subroutine which returns a subgraph W' of the collaboration graph W in which only edges with weights higher than or equal to t remain. As long as the subgraph W' is not empty, the inner **while** loop runs. It invokes the *ConnectedComponent* subroutine on W' to find a group of nodes u in W' that are all reachable from each other. If a group of cardinality greater than 1 is found, or if no edges had been filtered out of W to get W' in this current iteration, the newly discovered group of nodes u is allocated as a new unit, and the nodes in u are removed from both W and W' using the *RemoveNodes* subroutine. Otherwise, the nodes in u are not yet allocated as an unit, and are removed from W' but not from W; indeed, nodes in u will be allocated during subsequent iterations of the outer **while** loop with lower values of the collaboration threshold. When W' becomes empty, the inner **while** loop terminates, the current collaboration threshold t that was used is discarded from L, and the next iteration of the outer **while** loop continues with the next lower value in L as the new current collaboration threshold. Eventually, when W is empty, the outer **while** loop terminates, and the algorithm terminates. At that point, every function in S has been allocated to exactly one unit in the resulting set U of units.

Algorithm 4 uses the following subroutines:

1. The subroutine *CollaborationGraph* takes as input a set of (boolean) control interfaces of a set of functions S and creates a weighted undirected graph $W = (V, E)$. The graph is such that there is a vertex $v_f \in V$ for each function f in S, and an edge $(v_f, v_g) \in E$ of weight w for every pair of functions f and g in S such that w is the degree of collaboration between f and g.

2. The subroutine *NumberOfCollaborationClasses* takes as input a collaboration graph and returns a positive integer c based on the *collaboration classification policy*; the policy `sc`n forces the subroutine to always return the value n.

3. The subroutine *CollaborationThresholds* takes as inputs a collaboration graph W and an integer c, and returns a sequence of c distinct non-negative integers starting from 0 and dividing equally the interval between 0 and the maximum weight in W.

4. The subroutine *FilterLightEdges* takes a weighted undirected graph $W = (V, E)$ and returns a graph $W' = (V, E')$ such that $E' \subseteq E$, and for all edges $e \in E$ of weight w we have $e \in E'$ if and only if $w \geq c$.

5. The subroutines *IsEmpty* and *RemoveNodes* are defined as before.

6. The subroutine *ConnectedComponent* takes as input a weighted undirected graph $W' = (V, E)$ and returns a set $u \subseteq V$ of nodes. The set u is such that every pair of nodes $v_1, v_2 \in u$ are connected in W', and for all nodes $v_3 \in V$, if $v_3 \notin u$, then for all $v_1 \in u$, v_1 and v_3 are not connected.

Theorem 6.2 *For a call graph $\mathcal{G}$ over a set of functions S and given a set of control interfaces of S, the algorithm* PartitionSC(S) *creates a partition of $\mathcal{G}$.*

Proof. We prove (i) that the algorithm terminates, and (ii) that when it has terminated, (a) every function in S is allocated to some unit u generated by the algorithm, and (b) that no function $f \in S$ is allocated to two units u and u' generated by it.

The subroutine *CollaborationThresholds* returns a list of integers in which the last element is the maximum weight in the collaboration graph W. Since one element

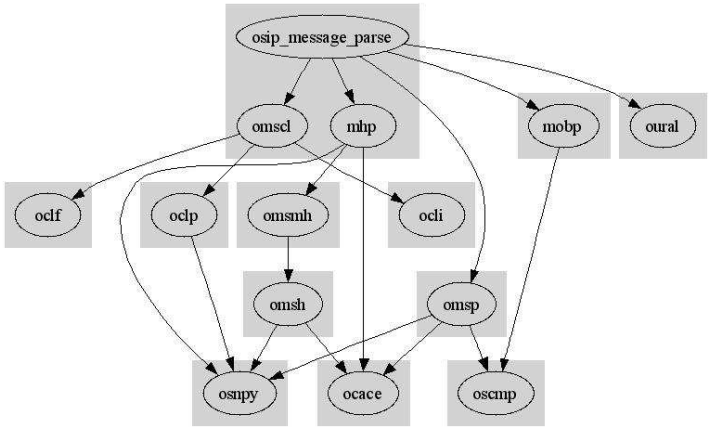


Figure 6.1. The call-graph, and the partition created by `sc7`

from that list is discarded when the inner **while** loop exits, and since the inner **while** loop is guaranteed to exit (since it removes at least one vertex from the finite graph W' in each iteration and exits when W' has no more vertices), the last element of the list L is eventually assigned to t (in line 5). Thus, the condition $W' = W$ in the **if** condition in the inner **while** loop is satisfied eventually. From that point onwards, at least one vertex is removed from the graphs W and W' in each iteration of the inner **while** loop. Since the graph W is finite, the outer **while** loop must eventually exit when W becomes empty. Thus, the algorithm terminates.

Since every function in S is a vertex in the collaboration graph W, and since the outer loop runs until the graph W is empty, and since a vertex is removed form W only if it is allocated to a generated unit, it follows that every function in S is allocated to some unit. For the part (ii)(b), whenever a function f is allocated to a unit u (in line 10 of the algorithm), it is next removed from both W and W', and thus cannot be reallocated later to another unit.

Figure 6.1 shows a small fragment of the `osip` call-graph, chosen for simplicity.

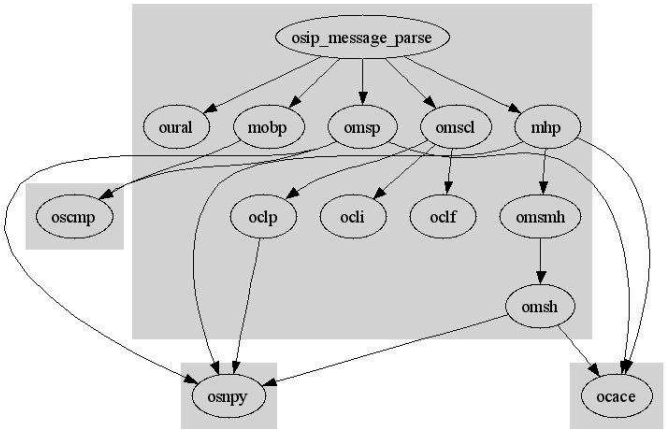


Figure 6.2. The partition created by `cp1i`

The shaded boxes show the partition created by running `sc7` on the graph. On the same graph, running `cp1i` partitions the functions as shown by Figure 6.2. Note that each of the functions in the big fragment have 1 or 0 callers, whereas `oscmp`, `osnpy`, and `ocace` are more generic, having multiple callers. Running `cp3` partitions the functions into two groups: {`osnpy`} and the rest; `osnpy` is the only function with more than three callers.

6.5 Experimental Results

We have implemented the algorithms presented in the previous section in a prototype tool for partitioning programs written in C. In order to evaluate the effectiveness of these algorithms, we applied our tool to a large[1] software application: oSIP, an open-source implementation of the *Session Initiation Protocol*. SIP is a telephony protocol for call-establishment of multi-media sessions over IP networks (including "Voice-over-IP"). oSIP is a C library available at

[1]From a unit testing and verification point of view.

`http://www.gnu.org/software/osip/osip.html`. The oSIP library (version 2.2.1) consists of about 30,000 lines of C code. Two typical applications are SIP clients (such as softphones to make calls over the internet from a PC) and servers (to route internet calls). SIP messages are transmitted as ASCII strings and a large part of the oSIP code is dedicated to parsing SIP messages.

Our experimental setup was as follows. We considered as our program P a part of the oSIP parser consisting of the function `osip_message_parse` and of all the other oSIP functions called (directly or indirectly) by it. The resulting program consists of 55 functions, which are described by several thousands lines of C code.[2] The call graph of this program is actually acyclic, which we believe is fairly common. (Note that our partitioning algorithms are not customized to take advantage of this property.) It is worth observing that the "popularity" (as defined earlier) distribution in this program is far from uniform: about half the functions (25 out of 55) are very unpopular, being called by only 1 or 2 callers, about 20 have around 5 callers each, and the remaining 8 are very popular, with 20 or more callers.

We used an extension of the DART implementation described in [83] to perform all our testing experiments. For each unit, the inputs controlled by DART were the arguments of the unique toplevel function of that unit, and DART was limited to running a maximum of 1,000 executions (tests) for testing each unit. In other words, if DART did not find any error within 1,000 runs while testing a unit, testing would then move on to the next unit, and so on. Since the oSIP code does not contain assertions, the search performed by DART was limited to finding segmentation faults (crashes). Whenever DART triggered such an error, the testing of the corresponding unit was stopped. Because of the large number of errors generated by all these

[2]The C files containing all these functions represent a total of 10,500 lines of code.

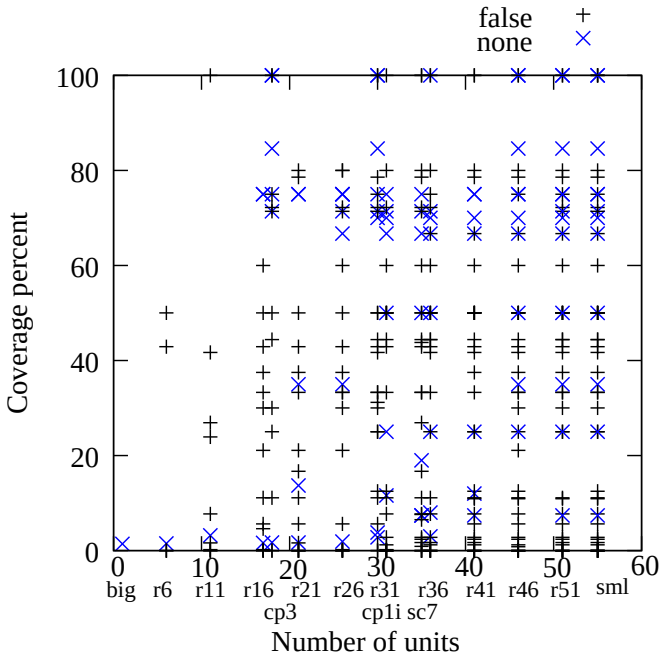


Figure 6.3. Coverage and incidences of false alarms

experiments (see below), we could not visually inspect each of those; we therefore assumed that the overall program could not crash on any of its inputs and hence that all the errors found by DART were spurious, i.e., false alarms. Thus, in this experimental setup, at most one false alarm can be reported per unit. Coverage is defined as branch coverage. Test coverage while testing a specific unit is defined and measured with respect to the code (branches) of that unit only.

We performed experiments with partitions generated by several partitioning algorithms: the symbol `cp1i` denotes the partition generated by *both* the partitioning algorithm that uses "callee popularity" iteratively *or* with a cutoff of 1, as both partitioning algorithms happened to generate the same partition for the oSIP code considered here; `cp3` represents the partition generated by "callee popularity" with a cutoff of 3; and `sc7` denotes the partition generated by the "shared code" partitioning algorithm with a policy value of 7. The above parameter values were chosen arbitrar-

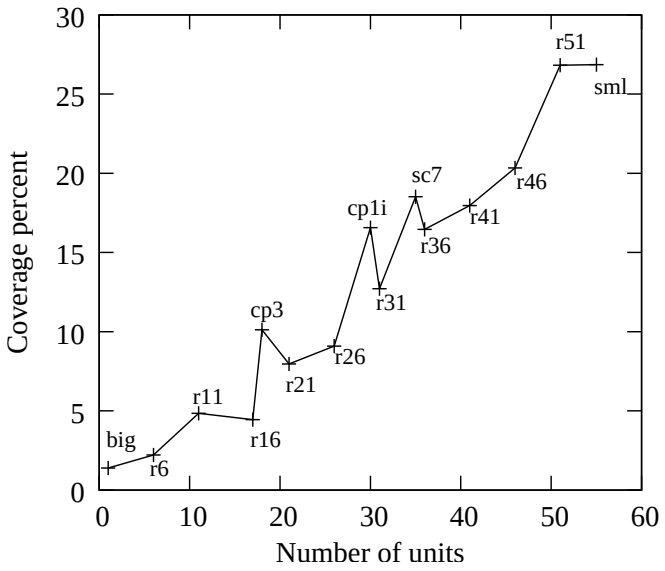


Figure 6.4. Overall branch coverage

ily. To calibrate our results, we also performed experiments with the extreme partition consisting of a single unit containing all the functions, denoted by `big`, and with the other extreme partition where each unit consists of a single function, denoted by `sml`. Finally, we also performed experiments with a set of randomly generated partitions, each denoted by `r`n where n is the number of units of the corresponding partition.

Our experimental results are shown in a series of figures. Figure 6.3 shows for each partition, the coverage obtained for each unit in the partition, and whether a false alarm was reported for that unit. A (blue) $\times$ mark indicates no false alarm was reported (i.e., the unit could be successfully tested 1,000 times with different inputs), while a (black) + mark indicates that DART reported a false alarm (i.e., found a way to crash the unit within a 1,000 runs). (Thus, for n units, there are n marks on the corresponding column, but some of these are superposed and not distinguishable.) All those experiments took about one day of runtime on a Pentium III 800Mhz processor running Linux. Note the low coverage of 1% in `big`; this indicates that monolithic

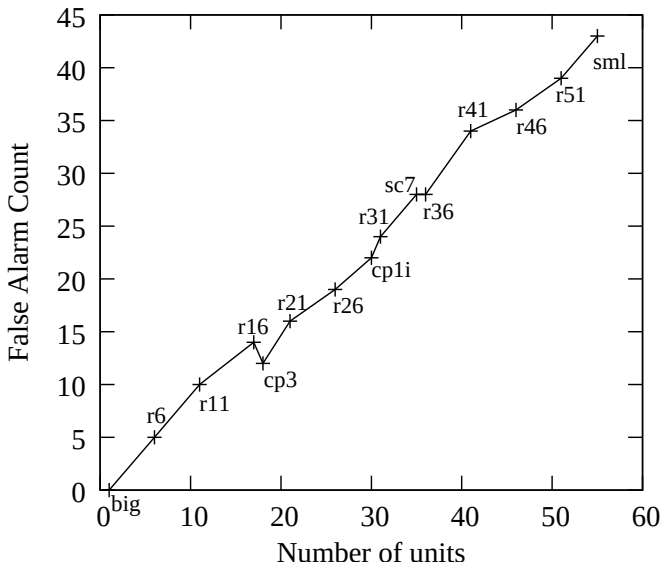


Figure 6.5. Number of false alarms

testing is severely ineffective. Also from this figure (and subsequent figures), it can be seen that the increase in coverage in the units of partitions to the right (which contain more units) is not free of cost since it also yields more false alarms.

For each partition considered, Figure 6.4 shows the overall branch coverage obtained after testing all the units in the corresponding partition. As is clear from Figure 6.4, the branch coverage in a partition tends to increase as the number of units in the partition rises and the average size of each unit in the partition gets smaller. At the extremes, coverage rises from about 1% in `big` to about 27% in `sml`.

These perhaps seemingly low coverage numbers can be explained as follows: the total number of program branches is large (1,162), not all the branches are executable in general or in our specific experimental setup –for instance, we do not inject errors for calls to `malloc`, `free`, etc. so branches that are executed in case of errors to calls to these functions are not executable–, testing of a unit stops as soon as an error is found or after 1,000 tests have been run, and finally, whenever DART cannot

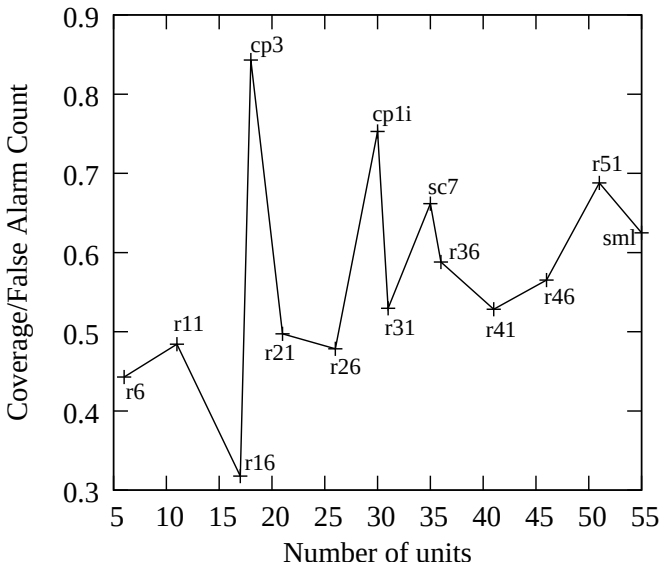


Figure 6.6. Ratio of coverage to false alarm

reason symbolically about some input constraint, it reduces to random testing [83] and random testing is known to usually yield very low code coverage [126, 83].

Note that the increase from `big` to `sml` is not monotonic: there are peaks corresponding to each of our partitioning algorithms `cp3`, `cp1i` and `sc7`. Thus, even though overall coverage rises as the number of units in the partition increases, our partitioning algorithms are able to select the units cleverly enough to raise the inter-unit cohesion sufficiently to consistently beat the overall coverage obtained by random partitions with similar numbers of units.

From Figure 6.4 it is clear that coverage on the whole rises as the number of units in a partition increases, and that `sml` is the best partition with respect to coverage. However, this higher coverage is obtained at the cost of an increased number of false alarms, as can be seen from Figure 6.5, which gives the absolute number of false alarms found for each partition.

We thus have a tension between two competing objectives: maximizing overall test

coverage while minimizing the number of false alarms. In order to evaluate how the different partitioning algorithms satisfy these two competing objectives, Figure 6.6 shows the ratio of the overall coverage obtained with a partition divided by the number of false alarms reported for the partition. (The value of this ratio is undefined for `big` as it has zero false alarms; however, `big` is not a serious contender for the title of best partitioning scheme as it leads to very low code coverage.) Observe that `cp3`, `cp1i`, and `sc7` each correspond to clear peaks in the graph, although of steadily decreasing magnitude, indicating that `cp3` is the best suited partitioning scheme among those considered for the specific code under analysis. These peaks mean that all three algorithms clearly beat neighboring random partitions, even though these three algorithms exploit only simple control interface definitions ignoring all data dependencies. Whether using more elaborate interface definitions like those discussed in Section 6.3 can significantly improve those results is left for future work.

6.6 Discussion and Other Related Work

Interfaces are defined in the previous sections as *syntactic* notions, which are computable via static analysis, and the complexity of interfaces is then used as a *heuristic* measure of how intertwined functions are. However, an ideal software partitioning algorithm would cut apart a function f calling another function g only if any input of g is *valid*, i.e., does not trigger a false alarm. For instance, if function g takes as input a boolean value, it is very likely that both 0 and 1 are valid inputs. In contrast, if g takes as argument a pointer or a complex data structure, its inputs are likely to be *constrained* (e.g., g may assume that an input pointer is non-null, or that the elements of its input array are sorted): g assumes that all its inputs satisfy a *precondition*.

Unfortunately, inferring preconditions automatically from program analysis (dubbed the *constraint inference problem* in [82]) is obviously as hard as program verification itself. Another strategy would be to perform piecemeal testing (see Section 6.2), then hide interfaces where false alarms were generated, and repeat the process until all false alarms have been eliminated. This strategy looks problematic in practice because it would require the user to examine all the errors generated to determine whether they are spurious or not, the number of iterations before reaching the optimum could be large, and the partition resulting from this process may often end up being the entire system, that is, no partition at all. Another impractical solution would be to generate and evaluate all possible partitions to determine which one is best. All these practical considerations explain why we seek in this work *easily-computable syntactic heuristics* that can generate an *almost optimal* partition at a *cheap cost*.

In spirit, our work is closest to work on automatically decomposing hardware combinatorial and sequential circuits into smaller pieces for compositional circuit synthesis, optimization, layout and automatic test-pattern generation (e.g., [87, 52, 127]). Circuit topology has also been exploited for defining heuristics for BDD variable orderings and for finding upper bounds on the sizes of BDD representations of combinatorial circuits (e.g., [24, 114]). However, the components, interfaces and clustering algorithms used for hardware decomposition are fairly different from those discussed in this chapter. Indeed, a digital hardware component is always finite state and its interface to the rest of the world is simply a set of pins (booleans). In contrast, defining exactly what the interface of a software component (say a C program) is already challenging (the flow of control between functions can be hard to determine, functions may take unbounded data structures as inputs, side-effects through global

variables are possible and sometimes hard to predict, etc.), and program verification for Turing-expressive languages is in general undecidable due to their possibly infinite state spaces.

Our work is also related to metrics for defining software complexity, such as *function points* (e.g., [76]) among others, and to work on *software architecture reconstruction* (e.g., [132]) that aim to facilitate the understanding of legacy code for reuse and refactoring purposes. Work on code reuse usually consider more sophisticated notions of "components" and "interfaces", which are often obtained through a combination of static, dynamic and manual analyses, but do not attempt to automatically partition code for facilitating a subsequent more precise automated program analysis.

Software partitioning for effective unit testing is related to compositional verification, which has been extensively discussed in the context of the verification of concurrent reactive systems in particular (e.g., see [84, 33, 55]). Compositional verification requires the user to identify components that can be verified in isolation and whose abstractions are then checked together. To the best of our knowledge, we are not aware of any work on heuristics for automatic software partitioning with the goal of compositional verification. The software partitioning heuristics presented in this work could also be used for suggesting good candidates of units suitable for compositional verification. However, it is worth emphasizing that the focus of this work has been (so far) the decomposition of *sequential* programs described by a set of functions, whose behaviors are typically more amenable to compositional reasoning than those of concurrent reactive systems, where compositional analysis (testing or verification) is arguably more challenging.

Algorithms for inter-procedural static analysis (e.g., [96, 36, 86]) and pushdown model checking (e.g., [35, 15]) are also compositional, in the sense that they can be

viewed as analyzing individual functions in isolation, summarizing the results of these analyses, and then using those summaries to perform a global analysis across function boundaries. In contrast, the software partitioning techniques we describe in this chapter are more light-weight since they provide only heuristics based on an analysis of function interfaces only (not the full function code), and since no summarization of unit testing nor any global analysis is performed. Software partitioning could actually be used to first decompose a very large program into units, which could then be analyzed individually using a more precise inter-procedural static analysis (since a single unit may contain more than one function). However, evaluating the effectiveness of a partitioning scheme when used in conjunction with static analysis tools (including static software model checkers like SLAM [23] or BLAST [93], for instance) would have to be done differently than in this chapter since (1) static analysis usually performs a for-all path analysis and hence does not measure code coverage as is done during testing, and (2) static analysis reports (typically many) false alarms due to abstraction and the imprecision that it always introduces, in addition to false alarms simply due to missing environment assumptions as with testing. Another interesting problem for future work (first suggested in [82]) is how to perform automatic dynamic test generation compositionally using a summarization process similar to what is done in inter-procedural static analysis.

Finally note that, although we used a DART implementation to perform the experiments reported in the previous section, the software partitioning problem and the techniques that we proposed to address it are independent of any particular automated test generation framework. We refer the reader to [83, 82] for a detailed discussion on other automated test generation techniques and tools.

6.7 Conclusion

We studied in this chapter how to automatically partition a large software program into smaller units that can be tested in isolation using automated test generation techniques without generating (too many) false alarms due to unrealistic inputs being injected at unit interfaces exposed by the partitioning. We proposed an approach that identifies control and data inter-dependencies between program functions using static analysis, and divides the source code into units where highly-intertwined functions are grouped together. We presented several partitioning algorithms based on this idea.

Preliminary experiments show that, perhaps surprisingly, *even partitioning algorithms exploiting only simple control dependencies can already significantly increase test coverage without generating too many false alarms*. These experiments also seem to validate the intuition behind these algorithms, namely that *grouping together highly-intertwined functions in the same unit improves the effectiveness of testing*, since those algorithms are able to consistently beat random partitions with similar numbers of units for our benchmark.

More experiments are needed to confirm these observations. Also we do not claim that our specific partitioning algorithms are the best possible: we have only shown that *there exist some simple partitioning algorithms that can beat random partitions*, but other partitioning algorithms (and parameter values) should be experimented with. Whether using more elaborate interface definitions like those discussed in Section 6.3 can improve those results is also left to be investigated.

Still, we believe our preliminary results are an encouraging *first step towards defining light-weight heuristics to partition large software applications* into smaller

units that are amenable to (otherwise *intractable*) more precise analyses, such as dynamic software model checking.

Acknowledgements

The work reported in this chapter was conducted jointly with Dr. Patrice Godefroid at Bell Laboratories, Lisle, IL. We thank Dr. Nils Klarlund for helpful comments on preliminary ideas that led to this work. This work was funded by Dr Godefroid supported in part by NSF CCR-0341658. This chapter is based on a paper [43] presented at EMSOFT 2006, copyright held by ACM[3], 2006; and Bell Laboratories Technical Memorandum ITD-06-46767J [44], co-authored with Dr Patrice Godefroid.

[3] `http://doi.acm.org/10.1145/1176887.1176925`

Algorithm 4 PartitionSC(S)

Input: A set of control interfaces of a set S of functions

Output: A partition of S into a set U of units

Variables: *WeightedGraph* W

1: $W :=$ *CollaborationGraph*(S)
2: $c :=$ *NumberOfCollaborationClasses*(G)
3: $L :=$ *CollaborationThresholds*(G, c)
4: **while** ($\neg$ *IsEmpty*(W)) **do**
5: $t := \max(L)$
6: $W' :=$ *FilterLightEdges*(W, t)
7: **while** ($\neg$*IsEmpty*(W')) **do**
8: $u :=$ *ConnectedComponent*(W')
9: **if** ($|u| > 1$ **or** $W' = W$) **then**
10: **add** u as a new unit in U
11: $W :=$ *RemoveNodes*(W, u)
end if
12: $W' :=$ *RemoveNodes*(W', u)
end while
13: $L := L \setminus \{t\}$
end while

Bibliography

[1] CHIC: Checker for interface compatibility. `http://www.eecs.berkeley.edu/~tah/Chic`.

[2] JBUILDER. `http://www.codegear.com/products/jbuilder`.

[3] PTOLEMY II: Heterogenous modeling and design. `http://ptolemy.berkeley.edu/ptolemyII/`.

[4] VALGRIND. `http://valgrind.org/`.

[5] Martin Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP '89)*, volume 372 of *Lecture Notes In Computer Science*, pages 1–17. Springer-Verlag, 1989.

[6] Marwan Abi-Antoun, Jonathan Aldrich, Nagi Nahas, Bradley Schmerl, and David Garlan. Differencing and Merging of Architectural Views. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 47–58. IEEE Computer Society, 2006.

[7] Samson Abramsky. Game Semantics for Programming Languages (Abstract). In *Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science (MFCS '97)*, volume 1295 of *Lecture Notes in Computer Science*, pages 3–4. Springer-Verlag, 1997.

[8] Samson Abramsky. Games in the semantics of programming languages. In *Proceedings of the 11th Amsterdam Colloquium*, pages 1–6. ILLC, Dept. of Philosophy, University of Amsterdam, 1997.

[9] Samson Abramsky, Simon J. Gay, and Rajagopal Nagarajan. A type-theoretic approach to deadlock-freedom of asynchronous systems. In *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software (TACS '97)*, volume 1281 of *Lecture Notes in Computer Science*, pages 295–320. Springer-Verlag, 1997.

[10] Samson Abramsky, Dan R. Ghica, Andrzej S. Murawski, and C.-H. Luke Ong. Applying Game Semantics to Compositional Software Modeling and Verification. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 421–435. Springer-Verlag, 2004.

[11] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full Abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.

[12] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[13] Robert B. Allen and David Garlan. Formalizing architectural conection. In *Proceedings of the 16th IEEE International Conference on Software Engineering (ICSE '94)*, pages 71–80, New York, 1994. IEEE Computer Society Press.

[14] Robert B. Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6:213–249, 1997.

[15] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas W. Reps, and Mihalis Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):786–818, July 2005.

[16] Rajeev Alur, Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Automating modular verification. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR '99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 82–97. Springer-Verlag, 1999.

[17] Rajeev Alur and Thomas A. Henzinger. Reactive Modules. *Formal Methods in System Design*, pages 7–48, 1999.

[18] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49:672–713, 2002.

[19] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating Refinement Relations. In *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR '98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer-Verlag, 1998.

[20] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. Mocha: Modularity in Model Checking. In *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer-Verlag, 1998.

[21] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of Interface Specifications for Java Classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 98–109, New York, NY, USA, 2005. ACM.

[22] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, 2001.

[23] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Proceedings of the 13th International Conference on Computer-Aided Verification (CAV '01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, France, July 2001. Springer-Verlag.

[24] C. Leonard Berman. Circuit Width, Register Allocation and Ordered Binary Decision Diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(8):1059–1066, August 1991.

[25] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web Service Interfaces. In *Proceedings of the 14th International World Wide Web Conference (WWW '05)*, pages 148–159. ACM, 2005.

[26] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. An Interface Formalism for Web Services. Technical Report MTC-REPORT-2007-002, School of Computer and Communication Sciences (IC), Ecole Polytechnique Fédérale de Lausanne (EPFL), December 2007.

[27] Dirk Beyer, Arindam Chakrabarti, Thomas A. Henzinger, and Sanjit A. Seshia. An Application of Web-Service Interfaces. In *Proceedings of the 2007 IEEE International Conference on Web Services (ICWS '07)*, pages 831–838. IEEE Computer Society, 2007.

[28] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating Tests from Counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, 2004.

[29] Andrea Bianco and Luca de Alfaro. Model Checking of Probabilistic and Nondeterministic Systems. In *Proceedings of the 15th International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–513. Springer-Verlag, 1995.

[30] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[31] Patricia Bouyer, Antoine Petit, and Denis Thérien. An algebraic approach to data languages and timed languages. *Information & Computation*, 182:137–162, 2003.

[32] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated Testing Based on Java Predicates. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*, pages 123–133. ACM Press, July 2002.

[33] Tevfik Bultan, Jeffrey Fischer, and Richard Gerber. Compositional Verification by Model Checking for Counter-Examples. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '96)*, pages 224–238. ACM Press, 1996.

[34] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[35] Olaf Burkart and Bernhard Steffen. Model Checking for Context-Free Processes. In *Proceedings of the 3rd International Conference on Concurrency Theory (CONCUR '92)*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer-Verlag, 1992.

[36] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience (SPE)*, 30(7):775–802, 2000.

[37] Cristian Cadar and Dawson R. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN '05)*, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23. Springer-Verlag, 2005.

[38] Arindam Chakrabarti. Interface Compatibility Checking for Software Modules. M.S. Report, University of California at Berkeley, Berkeley, CA, December 2005.

[39] Arindam Chakrabarti, Krishnendu Chatterjee, Thomas A. Henzinger, Orna Kupferman, and Rupak Majumdar. Verifying quantitative properties using bound functions. In *Proceedings of the 13th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*

(CHARME '05), volume 3725 of *Lecture Notes in Computer Science*, pages 50–64. Springer-Verlag, 2005.

[40] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, Marcin Jurdziński, and Freddy Y. C. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 428–441. Springer-Verlag, 2002.

[41] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Synchronous and Bidirectional Component Interfaces. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 414–427. Springer-Verlag, 2002.

[42] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Resource Interfaces. In *Proceedings of the Third International Conference on Embedded Software (EMSOFT '03)*, volume 2855 of *Lecture Notes in Computer Science*, pages 117–133. Springer-Verlag, 2003.

[43] Arindam Chakrabarti and Patrice Godefroid. Software Partitioning for Effective Automated Unit Testing. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT '06)*, pages 262–271. ACM Press, 2006.

[44] Arindam Chakrabarti and Patrice Godefroid. Software Partitioning for Effective Compositional Testing and Dynamic Model Checking. Technical Report ITD-06-46767J, Bell Laboratories, January 2006.

[45] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *CAV 96: Proc. of 8th Conf. on Computer Aided Verification*, volume 1102 of *Lect. Notes in Comp. Sci.*, pages 419–422. Springer-Verlag, 1996.

[46] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[47] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[48] Paul C. Clements, David Garlan, Reed Little, Robert L. Nord, and Judith A. Stafford. Documenting Software Architectures: Views and Beyond. In *ICSE*, pages 740–741, 2003.

[49] Christoph Csallner and Yannis Smaragdakis. Check'n Crash: Combining Static Checking and Testing. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 422–431. ACM Press, 2005.

[50] Julio Leao da Silva Jr., J. Shamberger, M. Josie Ammer, C. Guo, Suet-Fei Li, Rahul C. Shah, Tim Tuan, Michael Sheets, Jan M. Rabaey, Borivoje Nikolic, Alberto L. Sangiovanni-Vincentelli, and Paul K. Wright. Design Methodology for Picoradio Networks. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE '01)*, pages 314–323. IEEE Computer Society Press, 2001.

[51] Mads Dam. CTL* and ECTL* as Fragments of the Modal μ-calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[52] S. R. Das, Wen-Ben Jone, A. R. Nayak, and I. Choi. On Testing of Sequential Machines Using Circuit Decomposition and Stochastic Modeling. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(3):489–504, March 1995.

[53] Luca de Alfaro, Rajeev Alur, Radu Grosu, Thomas A. Henzinger, M. Kang, Rupak Majumdar, Freddy Y. C. Mang, Christoph Meyer-Kirsch, and Bow-Yaw Wang. Mocha: A Model Checking Tool that Exploits Design Structure. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, 2001.

[54] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *Proceedings of 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE '01)*, pages 109–120. ACM Press, 2001.

[55] Luca de Alfaro and Thomas A. Henzinger. Interface Theories for Component-Based Design. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT '01)*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer-Verlag, 2001.

[56] Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. From Verification to Control: Dynamic Programs for Omega-regular Objectives. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS '01)*, pages 279–290. IEEE Computer Society Press, 2001.

[57] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed Interfaces. In *Proceedings of the 2nd International Workshop on Embedded Software (EMSOFT '02)*, volume 2491 of *Lecture Notes in Computer Science*, pages 108–122. Springer-Verlag, 2002.

[58] Luca de Alfaro and Rupak Majumdar. Quantitative solution of concurrent games. *Journal of Computer & Systems Sciences*, 68:374–397, 2004.

[59] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69, 2001.

[60] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, CA, 1998.

[61] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1988.

[62] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, Linköping, October 1999.

[63] A. Ehrenfeucht and J. Mychielski. Positional strategies for mean payoff games. *International Journal of Game Theory*, 8:109–113, 1979.

[64] E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *Proc. 32nd IEEE Symp. Found. of Comp. Sci.*, pages 368–377. IEEE Computer Society Press, 1991.

[65] E.A. Emerson and C. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, 1986.

[66] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*. Usenix Association, 2000.

[67] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[68] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.

[69] J. Filar and K. Vrieze. *Competitive Markov Decision Processes*. Springer-Verlag, 1997.

[70] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In *Proceedings of the 11th European Symposium on Programming (ESOP '02)*, volume 2305 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2002.

[71] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[72] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Compatibility Verification for Web Service Choreography. In *Proc. ICWS*, pages 738–741. IEEE, 2004.

[73] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, 1999.

[74] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of Interacting BPEL Web Services. In *Proc. WWW*, pages 621–630. ACM, 2004.

[75] Xiang Fu, Tevfik Bultan, and Jianwen Su. WSAT: A Tool for Formal Analysis of Web Services. In *Proc. CAV*, LNCS 3114, pages 510–514. Springer, 2004.

[76] S. Furey. Why we should use function points. *IEEE Software*, 14(2), 1997.

[77] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[78] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[79] David Gay and Alexander Aiken. Language support for regions. In *PLDI*, pages 70–80, 2001.

[80] Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 74–90, London, UK, 1999. Springer-Verlag.

[81] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–186, Paris, January 1997. ACM Press.

[82] P. Godefroid and N. Klarlund. Software model checking: Searching for computations in the abstract or the concrete (invited paper). In *Proceedings of 5th International Conference on Integrated Formal Methods (IFM)*, volume 3771 of *Lect. Notes in Comp. Sci.*, pages 20–32, Eindhoven, November 2005. Springer-Verlag.

[83] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, Chicago, June 2005. ACM Press.

[84] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.

[85] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, pages 219–227, Grenoble, September 2000. IEEE Computer Society Press.

[86] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82, Berlin, June 2002. ACM Press.

[87] S. Hassoun and C. McCreary. Regularity extraction via clan-based structural circuit decomposition. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 414–419, San Jose, November 1999. ACM Press.

[88] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 125–138, Berkeley, January 1992.

[89] M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous pi-calculus (extended abstract). In U. Montanari, J. Rolim, and E. Welzl, editors, *Automata, Languages and Programming, 27th International Colloquium*, volume 1853 of *Lect. Notes in Comp. Sci.*, pages 415–427, Geneva, Switzerland, 9–15 July 2000. Springer-Verlag.

[90] T.A. Henzinger, S. Qadeer, S.K. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. In G. Gopalakrishnan and P. Windley, editors, *FMCAD 98: Formal Methods in Computer-aided Design*, Lecture Notes in Computer Science 1522, pages 421–432. Springer-Verlag, 1998.

[91] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The blast software verification system. In *SPIN*, pages 25–26, 2005.

[92] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive Interfaces. In *ESEC/SIGSOFT FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 31–40, New York, NY, USA, 2005. ACM.

[93] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of*

Programming Languages (POPL), pages 58–70, Portland, January 2002. ACM Press.

[94] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[95] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[96] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 104–115, New York, October 1995. ACM Press.

[97] M. Huth and M. Kwiatkowska. Quantitative analysis and model checking. In *LICS*, pages 111–122. IEEE, 1997.

[98] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, 2000.

[99] O. H. Ibarra, J. Su, Z. Dang, T. Bultan, and R. A. Kemmerer. Counter machines: Decidable properties and applications to verification problems. In *MFCS*, LNCS 1893, pages 426–435. Springer, 2000.

[100] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 27(2):264–313, 2005.

[101] J. C. King. Symbolic execution and program testing. *Journal of the ACM*, 19(7):385–394, 1976.

[102] B. Korel. A dynamic approach of test data generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990. IEEE Computer Society Press.

[103] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[104] Patrick Lam, Viktor Kuncak, and Martin C. Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Notices*, 39(3):46–55, 2004.

[105] E. Larson and T. Austin. High coverage detection of input-related security faults. In *Proceedings of 12th USENIX Security Symposium*, Washington D.C., August 2003.

[106] Edward A. Lee and Yuhong Xiong. System-level types for component-based design. In *EMSOFT*, pages 237–253, 2001.

[107] Insup Lee, Anna Philippou, and Oleg Sokolsky. Process algebraic modeling and analysis of power-aware real-time systems. *Computing and Control Engineering Journal*, 13(4):180–188, 2002.

[108] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[109] N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[110] N.A. Lynch and M. Tuttle. Hierarcical correctness proofs for distributed algorithms. In *Proc. of 6th ACM Symp. Princ. of Dist. Comp.*, pages 137–151, 1987.

[111] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[112] David Garlan Massimo Tivoli. Adaptor Synthesis for Protocol-Enhanced Component Based Architectures. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA '05)*, pages 276–277, 2005.

[113] A. McIver and C. Morgan. Games, probability, and the quantitative μ-calculus $q\mu$. In *LPAR 02: Logic Programming and Automated Reasoning*, volume 2514 of *Lecture Notes in Computer Science*, pages 292–310. Springer-Verlag, 2002.

[114] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[115] Bertrand Meyer. Design by Contract: The Eiffel Method. In *Proceedings of the 26th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS '98)*, page 446. IEEE Computer Society, 1998.

[116] R. Milner. An algebraic definition of simulation between programs. In *Proc. of Second Int. Joint Conf. on Artificial Intelligence*, pages 481–489. The British Computer Society, 1971.

[117] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[118] J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[119] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[120] Srini Narayanan and Sheila A. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW*, pages 77–88. ACM, 2002.

[121] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, Portland, January 2002. ACM Press.

[122] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *PADL*, pages 56–70, 2004.

[123] Hanno Nickau. Hereditarily Sequential Functionals. In *LFCS '94: Proceedings of the Third International Symposium on Logical Foundations of Computer Science*, pages 253–264, London, UK, 1994. Springer-Verlag.

[124] M. Núñez and I. Rodrígez. Pamr: A process algebra for the management of resources in concurrent systems. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proc. FORTE'01*, volume 22 of *IFIP Conference Proceedings*. Kluwer Academic Publishers, 2001.

[125] National Institute of Standards and Planning Report 02-3 Technology. *The economic impacts of inadequate infrastructure for software testing*. May 2002.

[126] J. Offutt and J. Hayes. A semantic model of program faults. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 195–200, San Diego, January 1996. ACM Press.

[127] M. R. Prasad, P. Chong, and K. Keutzer. Why is ATPG easy ? In *Proceedings of the 36th Design Automation Conference (DAC)*, pages 22–28, New Orleans, June 1999. ACM Press.

[128] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: a Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[129] R. Shah and J. M. Rabaey. Energy aware routing for low energy ad hoc sensor networks. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, pages 812–817. IEEE Communications Society Press, 2002.

[130] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *SSYM'01: Proceedings of the 10th USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[131] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[132] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Proceedings of the 4th IEEE/IFIP Working Conference on Software Architecture (WICSA)*, Oslo, June 2004. IEEE Computer Society Press.

[133] M.Y. Vardi. A temporal fixpoint calculus. In *Proceedings of the 15th Annual Symposium on Principles of Programming Languages*, pages 250–259. ACM Press, 1988.

[134] W. Visser, C. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Boston, July 2004. ACM Press.

[135] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[136] Guoyan Xie, Zhe Dang, Oscar H. Ibarra, and Pierluigi San Pietro. Dense Counter Machines and Verification Problems. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV '03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 93–105. Springer-Verlag, 2003.