11th Monterey Workshop

Software Engineering Tools:
Compatibility and Integration

October 4 – 6, 2004

Hosted at Vienna Austria
Table of Contents
11th Monterey Workshop Abstracts. 11(1). 3. ............................................................. 3


Monterey Workshop 2004
October 4-6

CUE Workshop 2004
October 7

Software Engineering Tools:
Compatibility and Integration

Sponsored by:
Army Research Office
National Science Foundation
EU funds under the CUE Initiative

Vienna, Austria
Program of

Monterey Workshop 2004

Software Engineering Tools: Compatibility and Integration

Sponsored by:

Army Research Office
National Science Foundation

October 4-6, 2004
Hotel Sauerhof
Vienna, Austria

Workshop Chairs:
Thomas Henzinger, Ecole Polytechnique Federale de Lausanne
Zohar Manna, Stanford University
CUE Workshop

Sponsored by:

EU funds under the CUE Initiative
National Science Foundation

October 7, 2004
Hotel Sauerhof
Vienna, Austria

Workshop Chair:
Tom Maibaum, King’s College, London
Motivation and Objectives

Software is the new physical infrastructure of the information age. It is fundamental to economic success, scientific and technical research and national security. Our current ability to construct the large and complex software systems demanded for continued economic and military success are inadequate.

A lack of tool integration has been recognized as a major impediment to the construction of complex computer systems, especially in the development of embedded systems, where the design of a single artifact may involve up to 50 different tools, including design, functional analysis, performance analysis, etc. tools. Tool interoperability is an important prerequisite for increasing software productivity and reliability.

The challenge is to create a platform that brings together the formal methods community and user community. Research assumptions should be constrained to facilitate the integration of such point solutions into a more complete and coupled overall engineering capability. This necessarily requires standardization at some levels, like we see in the hardware and middleware community, but can still allow a wide variety of representations and differences in expressive power at higher levels, thus catering to different application areas. The platform can support a variety of proof techniques, static and dynamic analyzers, type checkers, documentation tools, as well as lower level tools that support these techniques, such as decision procedures, constraint solvers, and SAT solvers.

The advantages of such a platform are many for both researchers and users: it provides researchers with exposure to practical case studies and the opportunity to have their methods and tools evaluated. It provides users with access to state-of-the-art methods without the added burden of extensive adaptation and training.
Monterey Workshop Chairs

Thomas Henzinger, Ecole Polytechnique Federale de Lausanne
Zohar Manna, Stanford University

Program Committee

Bruce Krogh – Carnegie Mellon University
Matt Dwyer – Kansas State University
Tom Maibaum – King’s College, London, UK
Martin Wirsing – Ludwig Maximilians University Munich, Germany
Carlo Ghezzi – Polytechnic of Milan, Italy
Henny Sipma – Stanford University
Kane Kim – University of California at Irvine
Carlos Delgado Kloos – University Carlos III of Madrid, Spain
Lori Clarke – University of Massachusetts
Olaf Owe – University of Oslo, Norway
Insup Lee – University of Pennsylvania
Ugo Montanari – University of Pisa, Italy
Wolfgang Pree – University of Salzburg, Austria

Local Arrangements

Hermann Kopetz – Vienna University of Technology
Sibylle Kuster – Vienna University of Technology
# Program of the Monterey Workshop

**Sunday, October 3**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:00–9:00 pm</td>
<td>Welcome Reception</td>
</tr>
</tbody>
</table>

**Monday, October 4**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00</td>
<td>Opening Remarks</td>
</tr>
<tr>
<td></td>
<td>David Hislop, Army Research Office</td>
</tr>
<tr>
<td>9:30</td>
<td>Commercial Tool Integration: The Reactis Experience</td>
</tr>
<tr>
<td></td>
<td>Rance Cleaveland, SUNY at Stony Brook, Reactive Systems Inc.</td>
</tr>
<tr>
<td>10:10</td>
<td>Discussion</td>
</tr>
<tr>
<td>10:30</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>10:50</td>
<td>Integrated Tools for Integrated Methods</td>
</tr>
<tr>
<td></td>
<td>Alan Wassyng, McMaster University, Canada</td>
</tr>
<tr>
<td>11:10</td>
<td>Post-mortem analysis of embedded systems: A case study</td>
</tr>
<tr>
<td></td>
<td>Purush Iyer, North Carolina State University</td>
</tr>
<tr>
<td>11:30</td>
<td>TTP-Tools – The tool set of the Time-Triggered Architecture</td>
</tr>
<tr>
<td></td>
<td>Stefan Poledna, TTTech, Vienna</td>
</tr>
<tr>
<td>11:50</td>
<td>Discussion</td>
</tr>
<tr>
<td>12:15</td>
<td>Lunch</td>
</tr>
</tbody>
</table>

**Session 2: Model Driven Architectures**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:30</td>
<td>Model-Integrated Computing</td>
</tr>
<tr>
<td></td>
<td>Janos Sztipanovits, Vanderbilt University</td>
</tr>
<tr>
<td>2:10</td>
<td>Architecture-based Methods and Tools for Software System Design</td>
</tr>
<tr>
<td></td>
<td>Tom Maibaum, King’s College, London</td>
</tr>
<tr>
<td>2:30</td>
<td>Discussion</td>
</tr>
<tr>
<td>3:00</td>
<td>Coffee break</td>
</tr>
<tr>
<td>3:20</td>
<td>Architecture-based Verification</td>
</tr>
<tr>
<td></td>
<td>Lori Clarke, University of Massachusetts</td>
</tr>
<tr>
<td>3:40</td>
<td>Evolution of Architecture Models</td>
</tr>
<tr>
<td></td>
<td>Bernhard Rumpe, Technical University of Braunschweig</td>
</tr>
<tr>
<td>4:00</td>
<td>Model driven development of multimedia applications</td>
</tr>
<tr>
<td></td>
<td>Heinrich Hussman, Andreas Pleuss Ludwig-Maximilians-Universitaet, Munich</td>
</tr>
<tr>
<td>4:20</td>
<td>Discussion</td>
</tr>
<tr>
<td>5:00</td>
<td>Adjourn</td>
</tr>
</tbody>
</table>

Evening program: All participants are invited by TTTech for wine tasting at the vineyard Urbanusschenke Familie Eitler
Habsburgerstr. 62 a, 2500 Baden, Tel. 02252/209521
<table>
<thead>
<tr>
<th>Time</th>
<th>Session 3: Component-based Systems and Interface Theories</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30</td>
<td><em>Interface-based Design</em></td>
</tr>
<tr>
<td></td>
<td>Tom Henzinger, Ecole Polytechnique Federale de Lausanne</td>
</tr>
<tr>
<td>9:10</td>
<td><em>Threading by Appointment</em></td>
</tr>
<tr>
<td></td>
<td>Christoph Kirsch, University of Salzburg</td>
</tr>
<tr>
<td>9:30</td>
<td><em>The Creol approach to open distributed systems</em></td>
</tr>
<tr>
<td></td>
<td>Olaf Owe, University of Oslo</td>
</tr>
<tr>
<td>9:50</td>
<td>Discussion</td>
</tr>
<tr>
<td>10:20</td>
<td>Coffee break</td>
</tr>
<tr>
<td>10:40</td>
<td><em>Issues in Integration of TMO Programming and Specification Tools with Basic Object-Oriented Distributed Software Engineering Tools</em></td>
</tr>
<tr>
<td></td>
<td>Kane Kim, University of California, Irvine</td>
</tr>
<tr>
<td>11:00</td>
<td><em>Compositional Periodic Interface for Real-Time Components</em></td>
</tr>
<tr>
<td></td>
<td>Insup Lee, University of Pennsylvania</td>
</tr>
<tr>
<td>11:20</td>
<td><em>Priority Systems</em></td>
</tr>
<tr>
<td></td>
<td>Joseph Sifakis, Verimag, Grenoble</td>
</tr>
<tr>
<td>11:40</td>
<td>Discussion</td>
</tr>
<tr>
<td>12:15</td>
<td>Lunch</td>
</tr>
<tr>
<td>1:30</td>
<td>Session 4: Software Analysis and Design Tools</td>
</tr>
<tr>
<td>1:30</td>
<td><em>Multiple Interoperating Program Analyses</em></td>
</tr>
<tr>
<td></td>
<td>Martin Rinard, MIT</td>
</tr>
<tr>
<td>2:10</td>
<td><em>Timing Prediction and Timing Predictability</em></td>
</tr>
<tr>
<td></td>
<td>Reinhard Wilhelm, University of Saarland, Saarbrucken</td>
</tr>
<tr>
<td>2:30</td>
<td>Discussion</td>
</tr>
<tr>
<td>3:00</td>
<td>Coffee break</td>
</tr>
<tr>
<td>3:20</td>
<td><em>Model Checking, Abstraction and Symbolic Execution for Software</em></td>
</tr>
<tr>
<td></td>
<td>Sriram Rajamani, Microsoft Research</td>
</tr>
<tr>
<td>4:00</td>
<td><em>Linking theories of concurrency</em></td>
</tr>
<tr>
<td></td>
<td>He Jifeng, UNU/IIST, Macau</td>
</tr>
<tr>
<td>4:20</td>
<td>Discussion</td>
</tr>
<tr>
<td>5:00</td>
<td>Adjourn</td>
</tr>
<tr>
<td>7 pm</td>
<td>Evening Program</td>
</tr>
</tbody>
</table>

**Evening Program**

7 pm Workshop Dinner
### Wednesday, October 6

#### Session 5: Open Tool Integration Frameworks
**Chair:** Purush Iyer

<table>
<thead>
<tr>
<th>Time</th>
<th>Title</th>
<th>Speaker(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30</td>
<td><em>Black-box versus White-box Integration: Alternative Integration Strategies</em></td>
<td>Shmuel Katz, Technion</td>
</tr>
<tr>
<td>9:10</td>
<td><em>The Interface is the Message</em></td>
<td>Shankar, SRI</td>
</tr>
<tr>
<td>9:50</td>
<td>Discussion</td>
<td></td>
</tr>
<tr>
<td>10:20</td>
<td>Coffee break</td>
<td></td>
</tr>
<tr>
<td>10:40</td>
<td>Verification on the Web of Mobile Systems</td>
<td>Ugo Montanari, University of Pisa</td>
</tr>
<tr>
<td>11:00</td>
<td><em>XML Language Composition and Integration of XML-Processing Tools</em></td>
<td>Tomasz Janowski, UNU/IIST, Macau</td>
</tr>
<tr>
<td>11:20</td>
<td><em>(Testing) Tool Integration via Theory Integration</em></td>
<td>Bernhard Aichernig, UNU/IIST, Macau</td>
</tr>
<tr>
<td>11:40</td>
<td>Discussion</td>
<td></td>
</tr>
<tr>
<td>12:15</td>
<td>Lunch</td>
<td></td>
</tr>
</tbody>
</table>

#### Session 6: Standardized Modeling Environments
**Chair:** Joseph Sifakis

<table>
<thead>
<tr>
<th>Time</th>
<th>Title</th>
<th>Speaker(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:30</td>
<td>Tool Integration Aspects in the Model-Driven Architecture</td>
<td>Gabor Karsai, Vanderbilt University</td>
</tr>
<tr>
<td>2:10</td>
<td>Simulink integration patterns and lessons learned from integrating</td>
<td>Wolfgang Pree, University of Salzburg</td>
</tr>
<tr>
<td>2:30</td>
<td>Giotto/TDL and Simulink</td>
<td></td>
</tr>
<tr>
<td>3:00</td>
<td>Coffee break</td>
<td></td>
</tr>
<tr>
<td>3:20</td>
<td>Integration of formal methods in MDA using the model bus</td>
<td>Xavier Blanc, University of Pierre &amp; Marie Curie, Paris</td>
</tr>
<tr>
<td>3:40</td>
<td>Basing a Modeling Environment on a General Purpose Theorem Prover</td>
<td>Myla Archer, Naval Research Lab</td>
</tr>
<tr>
<td>4:00</td>
<td>Toward a more Formal Basis for Middleware</td>
<td>Chris Gill, Washington University at St Louis</td>
</tr>
<tr>
<td>4:20</td>
<td>Discussion</td>
<td></td>
</tr>
<tr>
<td>4:50</td>
<td>Closing Remarks</td>
<td></td>
</tr>
</tbody>
</table>
Program of the CUE Workshop

<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00</td>
<td>Integrating Tools for Practical Software Analysis</td>
</tr>
<tr>
<td>9:45</td>
<td>Zohar Manna and Henny Sipma</td>
</tr>
<tr>
<td></td>
<td>Discussion</td>
</tr>
<tr>
<td>10:30</td>
<td>Coffee break</td>
</tr>
<tr>
<td>11:00</td>
<td>Panel discussion: Architectures for Tool Integration</td>
</tr>
<tr>
<td></td>
<td>Chair: Gianluigi Ferrari</td>
</tr>
<tr>
<td></td>
<td>Panel members: Joseph Sifakis, Bernhard Steffen, Rance Cleaveland, Lin Huimin</td>
</tr>
<tr>
<td>12:30</td>
<td>Lunch</td>
</tr>
<tr>
<td>14:00</td>
<td>Panel discussion: On the Need for Improved Methods, Models, and Languages in Tool-Based Software Development</td>
</tr>
<tr>
<td></td>
<td>Chair: Connie Heitmeyer</td>
</tr>
<tr>
<td></td>
<td>Panel members: Manfred Broy, Insup Lee, Joseph Sifakis, Alan Wassyng</td>
</tr>
<tr>
<td>15:30</td>
<td>Coffee break</td>
</tr>
<tr>
<td>16:00</td>
<td>Summary Discussion</td>
</tr>
<tr>
<td></td>
<td>Chair: Manfred Broy</td>
</tr>
<tr>
<td>17:00</td>
<td>Closing</td>
</tr>
<tr>
<td>Name</td>
<td>Institution</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Bernhard Aichering</td>
<td>United Nations University – IIST, Macau</td>
</tr>
<tr>
<td>Myla Archer</td>
<td>Naval Research Lab, Washington DC</td>
</tr>
<tr>
<td>Xavier Blanc</td>
<td>University of Pierre &amp; Marie Curie, Paris</td>
</tr>
<tr>
<td>Manfred Broy</td>
<td>Technical University Munich</td>
</tr>
<tr>
<td>Lori Clarke</td>
<td>University of Massachusetts, Boston, MA</td>
</tr>
<tr>
<td>Rance Cleaveland</td>
<td>State University of New York at Stony Brook and Reactive Systems, Inc</td>
</tr>
<tr>
<td>GianLuigi Ferrari</td>
<td>University of Pisa</td>
</tr>
<tr>
<td>Chris George</td>
<td>United Nations University – IIST, Macau</td>
</tr>
<tr>
<td>Chris Gill</td>
<td>Washington University at St Louis</td>
</tr>
<tr>
<td>Helen Gill</td>
<td>National Science Foundation, Washington DC</td>
</tr>
<tr>
<td>Klaus Havelund</td>
<td>Kestrel Technology / NASA Ames Research Center</td>
</tr>
<tr>
<td>He Jifeng</td>
<td>United Nations University – IIST, Macau</td>
</tr>
<tr>
<td>Connie Heitmeyer</td>
<td>Naval Research Lab, Washington DC</td>
</tr>
<tr>
<td>Tom Henzinger</td>
<td>Ecole Polytechnique Federale de Lausanne</td>
</tr>
<tr>
<td>David Hislop</td>
<td>Army Research Office</td>
</tr>
<tr>
<td>Heinrich Hussmann</td>
<td>Andreas Pleuss Ludwig-Maximilians-Universitaet, Munich</td>
</tr>
<tr>
<td>Purush Iyer</td>
<td>North Carolina State University, Raleigh, NC</td>
</tr>
<tr>
<td>Tomasz Janowski</td>
<td>United Nations University – IIST, Macau</td>
</tr>
<tr>
<td>Gabor Karsai</td>
<td>Vanderbilt University, Nashville, TN</td>
</tr>
<tr>
<td>Shmuel Katz</td>
<td>Technion, Haifa</td>
</tr>
<tr>
<td>Kane Kim</td>
<td>University of California at Irvine, CA</td>
</tr>
<tr>
<td>Christoph Kirsch</td>
<td>University of Salzburg</td>
</tr>
<tr>
<td>Hermann Kopetz</td>
<td>Vienna University of Technology</td>
</tr>
<tr>
<td>Fabrice Kordon</td>
<td>University of Pierre &amp; Marie Curie, Paris</td>
</tr>
<tr>
<td>Insup Lee</td>
<td>University of Pennsylvania</td>
</tr>
<tr>
<td>Lin Huimin</td>
<td>Institute of Software, Beijing</td>
</tr>
<tr>
<td>Tom Maibaum</td>
<td>King’s College, London</td>
</tr>
<tr>
<td>Zohar Manna</td>
<td>Stanford University, CA</td>
</tr>
<tr>
<td>Bertrand Meyer</td>
<td>Swiss Federal Institute of Technology, Zurich</td>
</tr>
<tr>
<td>Ugo Montanari</td>
<td>University of Pisa</td>
</tr>
<tr>
<td>Olaf Owe</td>
<td>University of Oslo</td>
</tr>
<tr>
<td>Name</td>
<td>Institution</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Klaus Pohl</td>
<td>University Duisburg, Essen</td>
</tr>
<tr>
<td>Stefan Poledna</td>
<td>TTTech, Vienna</td>
</tr>
<tr>
<td>Wolfgang Pree</td>
<td>University of Salzburg, Austria</td>
</tr>
<tr>
<td>Sriram Rajamani</td>
<td>Microsoft Research, Redmond, WA</td>
</tr>
<tr>
<td>Martin Rinard</td>
<td>MIT, Boston, MA</td>
</tr>
<tr>
<td>Bernhard Rumpe</td>
<td>Technical University of Braunschweig</td>
</tr>
<tr>
<td>Shankar</td>
<td>SRI, Menlo Park, CA</td>
</tr>
<tr>
<td>Josef Sifakis</td>
<td>Verimag, Grenoble</td>
</tr>
<tr>
<td>Henny Sipma</td>
<td>Stanford University, CA</td>
</tr>
<tr>
<td>Neeraj Suri</td>
<td>Technical University of Darmstadt</td>
</tr>
<tr>
<td>Janos Sztipanovits</td>
<td>Vanderbilt University, Nashville, TN</td>
</tr>
<tr>
<td>Alan Wassung</td>
<td>McMaster University, Canada</td>
</tr>
<tr>
<td>Reinhard Wilhelm</td>
<td>University of Saarland, Saarbrucken</td>
</tr>
</tbody>
</table>
Integrating Tools for Practical Software Analysis

Aaron R. Bradley  Henny B. Sipma
Sarah Solter  Zohar Manna

Computer Science Department
Stanford University
Stanford, CA 94305-9045
{arbrad,sipma,sbsolter,zm}@theory.stanford.edu

Abstract. The lack of integration between prototype implementations of results of research ("tools") blocks progress toward direct application of formal methods research in software engineering settings. We survey a host of tools, examining how their integration would increase their power and benefit future research and application. Based on this analysis, we describe a hypothetical and idealized Tool Integration Package (TIP). A TIP has two goals: first, it accelerates the research process by providing a range of tools in an integrated setting; and second, it serves as a ready-to-use tool for application in industry.

1 Introduction

Theoretically and practically, research in formal methods for software is closing in on the applicability barrier, that division between ivory tower research and research that gets noticed. But one problem remains: stand-alone tools. A significant portion of software that is released to support published research works in isolation. This isolation hampers future research efforts. Worse, the desired consumers — companies that produce software — are not interested. Even if an amazing new analysis is demonstrated to be an order of magnitude more accurate at a fraction of the computational cost, it almost certainly will be added to the growing number of implemented but unused research. Companies cannot waste resources on integrating a hundred prototype tools written in almost every language imaginable, only to have them produce a ream of disparate warnings, errors, proofs, and debug statements, 99% of which is pure speculation or superfluities. When we present them with one tool, they will listen.

Integrating tools is the next practical step in the development of our research. We will argue that a moderate effort now at integration will reward both the program analysis and the software engineering communities. To direct our look at tool integration, we will focus on these two groups — program analysis researchers (which includes researchers in, e.g., formal methods and compiler theory) and industrial software engineers. Of course, the domain of formal methods research is wide, encompassing not only software analysis, but also analysis of hardware and continuous, real-time, and hybrid systems, to name a few subdomains. Readability demands that we limit our scope, however, so we will focus
on software tools. The following two hypothetical situations motivate our argument.

*The researcher’s perspective* Suppose a researcher has developed the theory for a new analysis. It works well on handmade transition systems; it works well on the researcher’s favorite pseudo-language. Now he wants to test it on code. After choosing his favorite programming language, he faces several hurdles. First, he needs to parse the code. Programming languages are complex beasts, and parsing them was a research endeavor in itself a few decades ago. Clearly, writing his own parser is not an option. Second, even after solving the parsing problem, perhaps using a wonderful package like CIL [53], he realizes that extracting even rudimentary semantics from the parsed code is yet another research endeavor. Which references point to this piece of memory, or to this object? Which methods can be called here? On which type is this expression working? Is this path even feasible? What about this assertion — can it be trusted? There is no hope of implementing the analysis without relying on many other analyses (e.g., alias analysis, class hierarchy analysis, invariant generation, etc.) and foundational tools (e.g., decision procedures).

Assume he finally manages to write and run a few basic analyses, compile and use several different open source packages from fellow researchers, and compile and call a decision procedure package, having written a few wrapper API’s in the process, because some of the downloads were in a different language. Of course, this effort is not publishable and not funded, so it has to be done on the side. Hence, there is no time to do it properly and even less time to write documentation. While the researcher may in the end be able to do his analysis and publish a paper with impressive results, the effort that transferred his technology to real code will be largely wasted: his code will not be reusable. Other researchers have to go through the same process. Even the researcher himself will probably have to repeat it for future projects. Clearly, implementing one’s idea to work on real code is fraught with tedious tasks and misadventure. There must be a better way.

How can we make the process incremental, such that new analyses can be built on top of existing analyses, rather than each being a separate endeavor? We envisage the following scenario. The same researcher, who was struggling above, now has access to a TIP (specifically, a Tool Integration Package) for his favorite language, say $L_1$. It parses $L_1$, canonicalizing some of the convoluted aspects of $L_1$ and producing the intermediate language $L$, which is also produced by the TIPs for $L_2$ and $L_3$ (really the same TIP, just with different front end parsers). Common elements of the languages are factored out in $L$. He implements his method, but now has access to other analysis methods. For example, an alias analysis annotates variables of $L_1$ with identifiers to memory objects. A class hierarchy analysis narrows down the possible methods that can be called at a particular point. Another tool generates invariants, adding them as annotations to program points. The package also provides foundational tools such as SAT solvers and decision procedures, whose inclusion does not require more than a simple function call. Finally, to enable evaluation, the TIP comes with millions
of lines of open source code, classified by application area, complexity, and perhaps several more criteria, which can be used as the basis for experiments. If the method is an alternative to other published methods, this benchmark code provides a basis for comparison. These other published and implemented methods may already be available in the TIP, allowing the methods to be called in parallel and compared. If the new method is competitive, the researcher cleans up his code, documents it, provides API's for other analyses and submits it for inclusion in the TIP, where it will be available for others to use.

The practitioner’s perspective Looking at the other end of the ivory tower, we see software companies having problems of their own. In many software firms, the majority of software engineers are testers rather than developers. Of course, companies would like to change that ratio, increase their software’s robustness, and decrease their time-to-market. They do not look, however, to formal methods for answers. The main reason for ignoring the results of this research is that the methods and prototype tools seem irrelevant to their domain. They have not been demonstrated to apply to real software projects, are not known to scale, and perform isolated analyses. Companies simply cannot justify the effort of integrating these isolated tools in their software development process. There is no guarantee that even the next version of the same tool will not require yet another integration effort.

The availability of a TIP may help, especially if some standardization is respected. The TIP is a portal to transfer research results to practitioners. Build files may include selected analyses provided by the TIP, instructing the TIP to present relevant errors, warnings, speculations, and proofs in a single, readable, integrated summary. In the summary, the TIP displays warnings, speculations, and proofs hierarchically according to priority and confidence. Through this portal, researchers provide increasingly sophisticated analyses with improved accuracy, while the practitioners apply a seemingly stable tool via a solid front end. The software developers can focus on customization of input and output of the TIP, rather than having to worry about the correct integration of yet another version of yet another tool. From the software developer’s perspective, the TIP is simply an increasingly intelligent compiler.

Admittedly, this TIP oversimplifies the problem and solution. Nonetheless, it is clear that tool integration is a necessary next step for our field. In addition to practical integration of tools, this survey also examines possibilities for integration of fundamental methodologies. We feel that such activity is an unexplored mine of research opportunity.

Paper organization The survey is organized as follows. Section 2 starts by surveying a set of isolated tools for software analysis. The tools represent several classes of theoretical foundations, including type theory, dataflow analysis, model checking, and theorem proving. After summarizing their capabilities, we zoom out to examine their foundations, looking for opportunities for integration of
methodologies. In Sections 3, 4, and 5, we turn to tools that support integration. Language (Section 3) and parsing (Section 4) tools include languages for representing systems, frameworks for parsing code and writing new analyses, and packages for translating between input languages. Foundational tools (Section 5) offer support for deep analyses in the form of decision procedures and mathematical tools. Section 6 discusses some of our experiences with integrating tools. It also proposes ideas for future integration of language, parsing, and foundational tools. Having closed the book on integration issues for researchers, Section 7 turns to software engineers, their needs, existing software to address these needs, and suggestions for future software. Finally, we make a few closing observations and suggestions in Section 8.

1.1 Further Reading

This paper presents tools based on fundamental research. We briefly give some historical foundation for the tools. For a comprehensive presentation of compiler theory, including static analyses, see [51]. Model checking [17, 57] and symbolic model checking [11] and bounded model checking (BMC) [8] were originally directed at hardware verification, but are increasingly used for software verification. See [18] for a textbook presentation. Abstract interpretation [21] was introduced as a general framework for constructing and executing state-based analyses of software and hardware. In [43], we find the first instance of an abstract interpretation, while in [22] is presented the famous abstract domain of polyhedra. Deductive verification has deep roots; see [47] for a textbook presentation and bibliographical references.

Engler’s and Musuvathi’s experience with static analysis and software model checking [31] gives an interesting perspective on deep versus shallow analyses. Rutar, Almazan, and Foster [58] summarize their experience with five Java bug finding tools. Additionally, they describe a meta-tool for combining the output of bug finding tools. It uses statistical metrics to pinpoint suspicious Java classes, based on the output of multiple bug finding tools. In [42] Jackson and Rinard argue that in the future, static analyses will be model-driven, deep analyses.

2 Tools

2.1 A Sampling of Tools

This section provides a description of a sample of tools. The sample is by no means exhaustive; many great tools were missed. We summarize key points in Table 1.

Java Pathfinder. Java Pathfinder (JPF) [62] is an explicit state model checker for Java. The first version translated Java to Promela and applied the Spin model checker [39]; the second version implements a model checker specific to Java, including a new JVM. In addition to model checking techniques, including efficient
<table>
<thead>
<tr>
<th>Tool</th>
<th>Language</th>
<th>B/V</th>
<th>Analysis</th>
<th>Foundation</th>
<th>Interaction</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPF</td>
<td>Java/Bytecode</td>
<td>B/V</td>
<td>Deep</td>
<td>Explicit MC</td>
<td>Moderate</td>
<td>Yes</td>
</tr>
<tr>
<td>Bandera</td>
<td>Java</td>
<td>B/V</td>
<td>Deep</td>
<td>Abstraction, MC</td>
<td>Extensive</td>
<td>Yes</td>
</tr>
<tr>
<td>CMC</td>
<td>C/C++</td>
<td>B</td>
<td>Deep</td>
<td>Explicit MC</td>
<td>Moderate</td>
<td>No</td>
</tr>
<tr>
<td>CBMC</td>
<td>C/C++</td>
<td>B/V</td>
<td>Deep</td>
<td>Bounded MC</td>
<td>Minimal</td>
<td>Yes</td>
</tr>
<tr>
<td>BLAST</td>
<td>C</td>
<td>B/V</td>
<td>Deep</td>
<td>Predicate abstraction</td>
<td>Moderate</td>
<td>Yes</td>
</tr>
<tr>
<td>SLAM</td>
<td>C</td>
<td>B/V</td>
<td>Deep</td>
<td>Predicate abstraction</td>
<td>Moderate</td>
<td>No</td>
</tr>
<tr>
<td>MAGIC</td>
<td>C</td>
<td>B/V</td>
<td>Deep</td>
<td>Predicate abstraction</td>
<td>Moderate</td>
<td>Yes</td>
</tr>
<tr>
<td>Spin</td>
<td>Promela</td>
<td>B/V</td>
<td>Deep</td>
<td>Explicit MC</td>
<td>Minimal</td>
<td>Yes</td>
</tr>
<tr>
<td>SAL</td>
<td>Pseudo</td>
<td>B/V</td>
<td>Deep</td>
<td>MC</td>
<td>Minimal</td>
<td>Yes</td>
</tr>
<tr>
<td>FindBugs</td>
<td>Java</td>
<td>B</td>
<td>Shallow</td>
<td>Static Analysis</td>
<td>Minimal</td>
<td>Yes</td>
</tr>
<tr>
<td>PREFix</td>
<td>C/C++</td>
<td>B</td>
<td>Shallow</td>
<td>Static Analysis</td>
<td>Minimal</td>
<td>No</td>
</tr>
<tr>
<td>xgcc</td>
<td>C/C++</td>
<td>B</td>
<td>Shallow</td>
<td>Static Analysis</td>
<td>Minimal</td>
<td>No</td>
</tr>
<tr>
<td>ESP</td>
<td>C</td>
<td>B/V</td>
<td>Moderate</td>
<td>Static Analysis</td>
<td>Minimal</td>
<td>No</td>
</tr>
<tr>
<td>CQual</td>
<td>C</td>
<td>V</td>
<td>Moderate</td>
<td>Type checking</td>
<td>Moderate</td>
<td>Yes</td>
</tr>
<tr>
<td>Astrée</td>
<td>C</td>
<td>V</td>
<td>Deep</td>
<td>Abstract interpretation</td>
<td>Minimal</td>
<td>No</td>
</tr>
<tr>
<td>ESC</td>
<td>Java, Modula-3</td>
<td>B</td>
<td>Moderate</td>
<td>Theorem proving</td>
<td>Extensive</td>
<td>Yes</td>
</tr>
<tr>
<td>STeP</td>
<td>Pseudo</td>
<td>V</td>
<td>Deep</td>
<td>Theorem proving</td>
<td>Extensive</td>
<td>Yes</td>
</tr>
<tr>
<td>PVS</td>
<td>Pseudo</td>
<td>V</td>
<td>Deep</td>
<td>Theorem proving</td>
<td>Extensive</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1. Summary of tools. "MC" abbreviates "model checking." "B" indicates that the tool is *primarily* a bug-finder; "V" indicates that the tool is *primarily* a verifier. Analysis is a rough estimate of how "deep" the analysis is, based on, e.g., whether the analysis is more syntactic or semantic. Foundation is a succinct categorization of the main research ideas implemented. Interaction rates the amount of user interaction required to get results. Free indicates whether the tool is available for research use.

state representation, JPF also implements predicate abstraction via the generation of abstract programs, program slicing using Bandera [20], partial-order reduction, and runtime analyses for race and deadlock detection. Partial-order reduction uses static analyses to determine independence between statements in parallel threads. JPF represents an integration of much research.

Applying JPF requires defining an environment (in Java). Additionally, the user can supply invariants and assertions; for predicate abstraction, the user must also provide a set of predicates. The developers cite partial automation of environment construction as a future goal.

As JPF works on bytecode, rather than source, it can be applied to other languages for which bytecode compilers exist, including Eiffel, Ada, O’Caml, Scheme, and Prolog. Additionally, one would assume that integrating the techniques with those of, e.g., Bandera and Flex, would be relatively straightforward. Indeed, JPF already works with Bandera for some of its functionality. Finally, the developers note that, unlike with source analyzers, JPF is able to analyze compiled libraries and programs that use libraries.

**Bandera** Bandera [20] constructs finite-state models from Java source; these models are then model checked using Spin [39], SMV [59], or SAL [25]. Un-
like other work on translating Java to Promela, Bandera focuses heavily on minimizing the size of the generated finite model, mainly through abstraction. First, given the user-supplied property and annotations, it soundly slices the Java source, extracting an abstraction that is relevant to the property. Second, it applies user-supplied abstractions to reduce the cardinality of types; e.g., it could apply a “sign” abstraction to integers or an ItemInVector abstraction to a particular object and vector. Third, it produces, e.g., optimized Promela.

Because it requires extensive user-interaction to generate the abstraction, Bandera supplies an elegant GUI. Specifying properties is made intuitive through property templates. Additionally, it provides access to a library of abstraction modules.

The developers made extensibility one of the goals of the project. Thus, adding new analyses and techniques, as well as exploiting the existing functionality, should be relatively easy.

**BLAST** BLAST (Berkeley Lazy Abstraction Software Verification Tool) [38] verifies safety properties of C programs using predicate abstraction. It has been successfully applied for verifying properties of drivers.

BLAST represents an integration of tools. For example, it parses C via CIL and uses several decision procedures (e.g., Simplify [54] and CVC [60]). Several aspects of BLAST would obviously benefit from greater integration. For example, the user manual [15] cites aliasing as a “major source of complexity in the implementation” — complexity that could have been factored out given a TIP. Additionally, the generation of seed predicates and the extraction of refinement predicates could easily be exported so that BLAST could potentially benefit from other analyses.

**SLAM** Like BLAST, SLAM [5] verifies C programs via predicate abstraction. The BLAST and SLAM strategies differ; however, SLAM has also been successfully applied to verifying properties of drivers and other real applications.

**MAGIC** MAGIC (Modular Analysis of Programs in C) [14] relies on predicate abstraction, like BLAST and SLAM. However, it builds SAT formulas that assert weak simulation between a given labeled transition system (LTS) and a C procedure. Additionally, the developers stress compositionality in their method, claiming that this approach allows the tools to spend relatively more time in constructing an abstract model. Targeted software includes implementations of security and network protocols.

**CMC** CMC (C Model Checker) [52] uses efficient explicit state model checking to find bugs in network protocols implemented in C and C++. Network protocols are suitable applications for analysis, as they are event-driven. CMC exploits the event-driven architecture in several ways. Memory requirements are reduced by ignoring the stack and registers, which are preserved by event handlers. Atomic
transitions are easily defined as the event handlers. Finally, the protocol’s interactions with the network are defined by calls to the event handler. Nevertheless, CMC is fully implemented to work with other applications. The user supplies correctness properties beyond the usual set of domain independent properties (e.g., never accessing illegal memory) and assertions. Additionally, the user defines an environment, consisting of stubs for functions like malloc, and identifies the initialization functions and event handlers.

Like BLAST [38] and SLAM [5], CMC requires relatively little effort when applied to applications of the same kind. Once correctness properties and an environment are defined for a particular type of network protocol, for example, checking several implementations should be straightforward. However, applying it to a new domain requires some effort.

**CBMC** CBMC [16] exploits bounded model checking (BMC) to verify correctness of C programs. It translates C code into SAT by unrolling loops a finite number of times, translating the C to SSA (static single assignment) form, and then constructing two bit vector equations, one for the program and one for the property. The negation of the usual implication between program and property is then compiled into SAT. Dynamic arrays, via malloc, are handled with uninterpreted functions.

Targeted programs include embedded software and prototype simulations that will eventually be implemented as hardware. Hence, its success relies on the fact that loops in such software can often be soundly unrolled a finite number of times. In the latter class of programs, CBMC checks formal equivalence between the C simulation and a Verilog description of the hardware.

The developers also provide a GUI that presents a debugger-like interface. Counter examples are presented as traces that can be incrementally executed, just like in a debugger. The combination of an automatic analysis and a programmer-friendly interface makes this tool undoubtedly useful for development of its targeted programs.

**Spin** Spin [39] is an extensively-used explicit-state model checker. Its input language, Promela, is expressive, enabling recent efforts at using Spin to verify software (e.g., JPf1 [37], Bandera [20], JCAT [26]).

**SAL** At SRI, an ongoing effort aims at integrating finite state tools with their existing infinite state tools. SAL (Symbolic Analysis Laboratory) [25] consists of an expressive language, one that includes higher types, predicate subtypes, datatypes, integers, reals, and more; an API to core functionality; and four model checkers implemented with the core functionality. The “infinite bounded” model checker extends bounded model checking, compiling level $k$ into the language of ICS [33], a combination decision procedure that includes theories for ground linear arithmetic, uninterpreted functions with equality, arrays, and several other theories.
SAL’s language is intended to be an intermediate language, so it is defined in XML. Functionality for parsing and pretty-printing SAL instances support its use. The developers plan to support translators from other modeling languages. Long term, the developers intend to integrate SAL with PVS [55] via specification translation. They also propose a “SAL Tool Bus,” through which other tools and SAL may interact.

FindBugs  FindBugs [40] looks for bugs in Java code based on bug patterns. Its core functionality rests on several standard dataflow analyses, as well as class hierarchy analysis. While its bug-finding techniques are simple pattern matches, the authors believe this shallow method is an effective approach, as many bugs in ordinary code arise from simple errors. For example, it is easy to mistakenly compare two Strings with the == operator instead of calling the equals() method — and easy to catch this mistake — but the error can lead to serious bugs in the program. FindBugs has the advantage that it requires low overhead on the part of the programmer: no annotations are needed, and both a GUI and command-line version are available. Additionally, it is possible to obtain it as a plug-in for Eclipse and as a task for the Apache Ant build tool.

PREFix  The development of PREFix [13] was motivated by the usability of Purify [41]; however, the developers wanted a static analyzer rather than a dynamic one. PREFix aims to identify many bugs, minimize false errors, and perform with little or no user interaction. Their reported results include statically identifying uninitialized memory; dereferences of NULL, uninitialized, or invalid pointers; and leaked memory, on software as large as Mozilla.

PREFix automatically constructs models of functions via symbolic simulation of paths. Models are essentially sets of guarded commands. A path includes assignments, guards for control flow, and descent into function calls. To mitigate effects of the possibly exponential number of paths, especially for root or near-root functions, the user may set an upper limit on the number of paths to explore. When a function is called, its model is used to evaluate its effect. Manually-constructed models are supplied for system calls; however, PREFix can run even without them, which allows it to handle library calls.

Metacompilation  Metacompilation (e.g., [36]) proposes a new perspective on finding bugs. Rather than finding bugs through failed proofs (either failed verification conditions or a counterexample produced by a model checker), a metacompiler lets the user specify rules that the source code should obey. In their simplest form, rules are state machines in which guards are based on state and patterns on AST nodes. The metacompiler then applies the rules in a context- and flow-sensitive analysis. Rules need not be sound; indeed, the useful ones are not. Rather, a good rule finds as many bugs as possible without raising too many spurious errors. The technique is enhanced by using weak decision procedures. Some assignments and guards are recorded in a constraint database.
during a depth-first exploration of a path; guards are then evaluated based on this constraint database, sometimes ruling out paths.

Extensions are specified via the Metal language; xgcc is the analysis tool. Extensions can range from simple state machines to complex analyses with C code. For example, subsequent work introduced statistical analyses to locate bugs [30]. Apparently, the technique is successful enough to support a company (http://www.coverity.com).

ESP  ESP (Error Detection via Scalable Program Analysis) [24] uses techniques similar to those of metacompilation, except that its analyses are sound. That is, if no errors are reported, then no errors exist. Further, it introduces a new type of data flow analysis that takes advantage of the states of the property automaton. In path-sensitive analysis, a potentially exponential number of paths are analyzed, sometimes resulting in exponential space requirements. Conversely, standard data flow analysis is not sensitive to paths, thus, for example, resulting in spurious error reports. Property simulation maintains execution state and property state as its data flow state; only data flow states with equal property states are merged. In a sense, only property-relevant path-sensitive information is maintained. This new analysis has polynomial time complexity. Using this technique, ESP has verified properties of large programs.

CQual  CQual [34] exploits type checking and type inference to verify deeper properties of C programs. Programmers add type qualifiers (e.g., const, tainted, and sorted) to the program text in the same way that C/C++ programmers currently use const. Type inference is then equivalent to verification. The advantages of the approach, relative to other annotation systems, is that programmers are already comfortable with using types. Specifying deeper properties via qualified types is thus a natural extension. Additionally, because CQual also infers qualifiers, programmers need not qualify every type. For example, adding qualifiers to library entry points may be enough to check certain properties. Finally, type checking and inference is decidable and efficient, so the method scales to large programs.

CQual supports implementing new qualifier-based analyses. The techniques in CQual complement other verification efforts. For example, inferring qualified types may strengthen a subsequent theorem proving effort of deeper properties. Qualified types could also serve as bases for predicate abstraction.

Astrée  Astrée (Analyseur statique de logiciels temps-réal embarqués) [48] exploits the abstract interpretation framework to verify the nonexistence of runtime errors in a restricted class of C programs. Runtime errors include out-of-bound array accesses, integer division by zero, floating point errors, unwanted integer arithmetic mod behavior, casts to smaller types, and user-provided asserts. The tool targets synchronous C programs without dynamic memory allocation, string manipulation, or complex pointer operations. One goal is to produce as
few spurious errors as possible in a realistic amount of time and memory. Astërée achieves impressive results, mainly through a wise choice of inexpensive abstract domains, smart heuristics, and (most likely) a well-written implementation.

ESC ESC (Extended Static Checking) for Modula-3 [27] and Java [45] applies automated theorem proving to find bugs statically. While ESC can be applied without any annotations — to find possible null-dereferences, for example — it is more powerful when the programmer provides method annotations in the form of preconditions, postconditions, and modify statements. For multithreaded programs, the programmer can supply annotations for “locking-level verification” in the form of declarations of which locks (are intended to) protect which variables and which locks can or should be held at program points. Annotations can even declare abstract variables, such as valid and state variables for the valid/state paradigm. The developers note, though, that annotations are often not required for client programmers; only the interface and implementation of a library need be annotated to catch many errors.

ESC is fundamentally based on generating (based on the annotations) and automatically proving verification conditions. Proofs come via the Simplify [54] decision procedure. More importantly, ESC extracts reasons for failure from failed proofs, thus identifying and explaining potential bugs.

STeP STeP (Stanford Temporal Prover) [9] is an interactive system for proving temporal properties, specified via LTL, of concurrent and real-time systems. The techniques are based on reducing the proof of temporal properties to proofs of first-order verification conditions, many of which the combination decision procedure automatically proves [47]. The system also provides a GUI for constructing and proving verification diagrams [46], algorithms for automatic invariant generation, and a model checker.

PVS PVS (Prototype Verification System) [55] presents a dynamic environment for specifying and verifying models. It includes a powerful combination of decision procedures and theorem proving capabilities, including induction, along with convenient support for defining “tactics.” While PVS is essentially a monolithic and isolated tool, its expressive specification language admits compilation from other research tools, facilitating a loose integration. PVS is not suitable by itself for analyzing large amounts of code, as it is primarily concerned with deep analyses of complex models.

2.2 Tool Foundations

An analysis can be classified as “shallow” or “deep,” depending on the degree to which it considers the semantics of programs — i.e., the behavior that the program text encodes. This loose characterization already suggests the variety of analysis techniques. In this section, we identify three broad categories of analyses based on three perspectives on programs and programming languages.
First, several tools rely on classical static analyses from compiler theory, such as type inference, alias analysis, class hierarchy analysis, and control flow graph construction. Type inference and dataflow analyses exploit the semantics of programming languages to discover facts about programs. Because these analyses were originally proposed and developed for compilers, they are fast — of low polynomial order — and sound. We call this semantic approach the classical perspective. CQual extends this perspective by coupling deeper program meaning with language semantics.

Analyses in the second category look at program semantics rather than programming language semantics. A program’s semantics is its reachable state space. We call this view the state-space perspective. Analyses following this trend focus on languages of computations, sets of (fair) sequences of program states. Properties of programs are reduced to questions of language inclusion, intersection, emptiness, etc., by transforming properties into languages — more sets of computations. Tools like ESC, STeP, and PVS handle infinite state spaces through theorem proving, mapping a program’s syntax onto its semantics via verification conditions. Tools like Bandera, Java Pathfinder, SLAM, and BLAST handle infinite state spaces by abstracting the state space, which they accomplish by abstracting the program. Theorem proving plays a role in generating and proving the soundness of the abstraction. Then model checking explores the abstract state space. Analyses that take the state-space perspective are “deep,” in that they explore the actual meaning of programs — their behavior within an environment. The syntax itself is mere description. These analyses are computationally expensive.

The third category, represented by PREFix, Metal and xgcc, and ESP, exploits the language perspective. Programs represent languages — sets of sequences of program statements — much the way that regular expressions represent regular languages. Of course, this parallel overlooks complicating factors, such as branch guards (which prune infeasible sequences), recursion, and aliasing (so that matching variables syntactically is unsound). Properties are merely specifications of other languages — other sets of sequences of program statements. This perspective is useful to the extent that behavioral properties can be mapped onto such languages. Consider, for example, the multiple-freeing example of [36]. The behavioral property specifies that no allocated memory should be freed more than once. The language property roughly specifies a language in which \texttt{free} is not applied more than once to any memory allocation point on any path through the program. Unsound containment checking of the analyzed program and the property reveals possible paths that are contained within the language of the program but not the language property. ESP [24] essentially performs a sound containment check so that it verifies (language) properties.

Mapping behavioral properties to pure language properties is the first way of exploiting the language perspective. Adding decision procedures and alias analyses strengthens the mapping between program syntax and program behavior. Online decision procedures allow the analysis to record (a subset of) assignments and branch decisions. The analysis can then identify some infeas-
ble paths via contradictions. From the language perspective, regarding branches as nondeterministic choices produces a language—a set of program paths, or sequences of statements—that overapproximates the set of feasible sequences. Soundly applying decision procedures to prune paths refines this overapproximation. Including an alias analysis strengthens the connection with program state. Properties may refer to memory allocation points rather than just to variable names. Finer alias analyses allow finer property checks (fewer false errors).

How can these categories and perspectives be integrated? Beyond integrating tools, we desire to integrate methodologies. Already, loose couplings are present in tools. xgcc, for example, uses decision procedures to exclude program paths. But let us consider this question in greater depth via two examples.

First, the analyses that take the classical perspective efficiently produce facts about programs. Such (relatively) easily obtained facts should strengthen abstraction and theorem proving efforts of state-space tools. Consider ESC. In Java, Collection classes store Objects, so retrieving Objects usually requires a dynamic cast. ESC flags such casts as possibly erroneous. However, suppose one supplements the ESC analysis with facts derived from a “Collection analysis.” Such an analysis might first apply an alias analysis and class hierarchy analysis (CHA) to form a sound overapproximation to the control-flow graph (CFG). The alias analysis would also allow the tracking of heap allocation points for Collection objects. The Collection analysis would associate with each such Collection object a set of types. If, during the analysis phase, it is discovered that an object of (static) type X may be added to a Collection object, X (and, implicitly, its subtypes) is added to the type set associated with each possible heap allocation point corresponding to the Collection object. Finally, this information could be accessed during the proof of verification conditions. Given that dynamic casts are usually correct, this combined analysis would probably eliminate most spurious error reports. Type qualifiers, too, could produce useful facts for theorem proving, as well as predicates for abstraction.

Second, the language, automata, and logic theories that support the state-space perspective should be exploited by the language perspective. In the language perspective, programs are in a sense finite state transition systems, albeit with many transitions and some caveats (if a state is a program statement, then a transition consists of a location guard and a nondeterministic choice of changing the current statement to one of its successors). Recognizing that correspondence, the logical next step is to apply, for example, full LTL/CTL/etc., model checking. Consider, for example, the following two properties, specifying (roughly) mutual exclusion and that no thread will retain forever a lock:

\[
(\forall \text{shared}_{\text{vars}} v)(\exists \text{mutex } l) \\
\square (\text{read}(v) \lor \text{write}(v) \rightarrow \neg \text{unlock}(l) \land \text{lock}(l))
\]

\[
(\forall \text{mutex } l)(\square \text{lock}(l) \rightarrow \Diamond \text{unlock}(l))
\]

shared_vars and mutex are suitably defined macros. These examples have a few motivating subtleties. First, in xgcc and ESP, property automata are implicitly
universally quantified. From one view, quantifier-free automata instances are generated and killed dynamically, each tracking, e.g., a variable instance. From another view, a universally quantified automaton tracks sets of, e.g., variable instances. In both views, only universal quantification is permitted, which full model checking reveals as unnecessarily restrictive.

Next, the second property is a progress property. Evaluation of the formula over a program would reveal that some loops must be proved terminating. This requirement is intuitive: a nonterminating loop between the lock and unlock statements could be dangerous. Thus, applying full model checking opens the door for practical application of recent work in termination of program loops. Property automata of xgcc and ESP implicitly specify safety properties. Third, properties may take the form of temporal formulas or, e.g., Büchi automata, whichever is more efficient for evaluation. Finally, we note that implementations are independent of property specification: path-sensitive and path-insensitive approaches both have merits and problems. For example, a path-insensitive LTL dataflow analysis, with conjunction for joining, is similar to the property simulation of [24], albeit with greater expressiveness. For this level of analysis, an online context-sensitive alias analysis (so that allocation points may be distinguished by call context rather than just program location) is necessary. Additionally, both sound and unsound implementations would have value.

Other opportunities for integration are clear. Many deep analyses, such as deductive verification, invariant generation, and termination analysis, traditionally work over friendly pseudo languages. To some extent, these analyses immediately become useful in the presence of a sound abstraction tool. For example, given a loop of C, an alias context (an overapproximation of what points to what), and a CFG, the abstracter might produce a sound number abstraction describing how the number variables of the loop evolve. Such an abstraction is then suitable for (number-based) termination analyses and (number-based) invariant generation. Here, classical analyses boost the effectiveness of state-space analyses. Going another way, generated invariants may aid in pruning paths in depth-first language analyses.

Clearly, integration of methodologies is fertile ground for future research. Because researchers with various backgrounds will need to become acquainted with tools and techniques from other domains, it is imperative that we put effort into integrating tools, like parsers, decision procedures, and math packages. In the next few sections, we consider existing tools for integrating implementations via common input representations, parsing data structures, and APIs.

3 Integration via Common Input Representations

Language-based integration allows tools to work together via input representations. As many tools work on various representation languages, efforts at integration focus on translation of one source representation into another. Such translations are difficult, as properties proved by one tool must be translated to the other tool and translated model — and still hold on the translated model.
For hardware and model analysis, such integration via translation is a somewhat *ad hoc* solution to the lack of a standard language. But for software analysis, language-based integration may offer a means of representing multiple languages (*e.g.*, C, C++, and Java) via a common overly expressive intermediate language. Rather than merely translating source, the common framework would export this common representation for direct analysis.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Language(s)</th>
<th>Purpose</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>VeriTech</td>
<td>CDL, SMV, Murphi, Spin, STeP, Petri nets</td>
<td>Translator</td>
<td>Yes</td>
</tr>
<tr>
<td>MathML</td>
<td>MathML</td>
<td>Common Representation</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Table 2.** Summary of language tools.

**VeriTech** One model for tool integration is based on translation of model representations and specifications. VeriTech [35] is a framework for writing source-to-source compilers. It defines an expressive core design language (CDL) to which source is compiled and from which new source is generated. VeriTech includes translators for SMV [59], Murphi [28], Spin [39], STeP [9], and Petri nets [56]. In some sense, the effort is like SAL; however, VeriTech does not include any analysis tools, as its sole objective is integration of other tools.

The developers of VeriTech have identified several nontrivial issues with source-to-source translation, mainly revolving around differing semantics. As translation often involves introducing new variables and transitions, the compiler records the correspondence between old and new variables and transitions. Furthermore, the developers formalize the *faithfulness* of translations and properties, defining when a property that holds in one source model holds in the other.

The VeriTech model of integration mainly focuses on hardware and model analysis tools, which vary widely in their languages and semantics. Programming languages like C, C++, and Java have more in common; however, semantic differences arising from, *e.g.*, typing, casting, and aliasing, present problems for compilation into a common intermediate language. The VeriTech effort might offer some solutions.

**MathML** While unrelated to program analysis, the W3C standardization effort, MathML [49], is worth mentioning. MathML is an XML-based language for representing mathematics. It facilitates both the display of mathematics and the maintaining of semantic information, for transfer between applications. Because parsers already exist, MathML is a candidate for saving and communicating the results of mathematically-intense program analyses, such as invariant generation.
4 Integration via Programming Language Parsers

Programming language parsers facilitate efforts at applying one’s research to real code. Parsers parse all or a significant subset of a language into an intermediate representation, such as an AST, three- or four-address code, or Java bytecode. Some tools apply convenient transformations to simplify code and remove language idiosyncrasies, such as moving side-effects from branch guards into the code blocks. Some parsers also allow code transformations, which allow the effects of analyses to propagate to other analyses.

Most parsers have been written with code optimization in mind. For code verification and bug finding, several features are desirable. General support for annotating code and querying other analyses would be useful. Currently, analyses that gather information useful to other analyses present the information via explicit queries. A complex analysis system will require an architecture for discovering and querying registered analyses. Further, some information is better presented as code annotations, information associated with, e.g., AST nodes. Such information includes inferred type qualifiers and automatically generated invariants. In general, whenever several analyses may supply information of the same type, such as invariants, it would be easier to process annotated code than to query all analyses. Such a structure makes analyses more flexible to changes and additions to the system’s registered analyses. The parsing system should also enforce standards of information presentation to maintain this flexibility. Finally, to support incremental analysis, the parser should be able to write and read annotated code, perhaps as XML.

Several other features would be helpful. The parser could be extensible so that techniques relying on programmer annotated code can read and manipulate the annotations within the parsing framework. Additionally, a parser and its associated decision procedures could share the same expression data structures, more closely integrating reasoning with parsing. No doubt, many other features are possible. Below we give a sampling of existing parsers.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Language</th>
<th>Representation</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL</td>
<td>C</td>
<td>AST</td>
<td>Yes</td>
</tr>
<tr>
<td>Flex</td>
<td>Java</td>
<td>Quadruples, Bytecode</td>
<td>Yes</td>
</tr>
<tr>
<td>Soot</td>
<td>Java Bytecode</td>
<td>Bytecode</td>
<td>Yes</td>
</tr>
<tr>
<td>Barat</td>
<td>Java</td>
<td>AST</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3. Summary of parsing tools.

CIL  CIL (C Intermediate Language) [53] parses C, producing an AST representation. In the process, it simplifies many unusual C constructs, easing the burden on the client. Moreover, some of its packaged analyses provide extra levels of simplification (e.g., translation to more CFG-like structure and translation to three-address code), which simplifies writing some analyses.
Writing analyses in CIL is remarkably easy. Written in O’Caml, CIL provides an intuitive API for visiting and transforming code. Transforming the code via manipulations of the AST allows analyses to affect each other: executing a set of code transforming analyses in sequence allows each analysis to benefit from the previously executed ones. Additionally, the included points-to analysis offers an example of information transfer via explicit queries. The data structure is first initialized as an analysis; the information is then available to other analyses for querying. Several desirable features are still missing from CIL, including a facility for annotating AST nodes. Annotations would provide a means of recording and transferring information such as invariants, proofs, inferred extended types, and held locks, without resorting to explicit queries. Nevertheless CIL forms an important first step toward a TIP; it certainly provides the most labor-intensive component of a TIP for C.

**Flex** Flex [32] is a Java compilation and program analysis framework. The Flex intermediate representation is based on quadruples. Many research efforts, focusing largely on program optimization and real-time Java, have used Flex for their implementation.

**Soot** Soot [61] is a Java compilation and optimization framework. Soot can represent Java bytecode in four ways: as streamlined bytecode (Baf), as three-address code (Jimple), as SSA three-address code (Shimple), and as three-address code suitable for code inspection (Grimp).

**Barat** Barat [10] parses Java source code into an AST representation. It supports traversals of the AST using the visitor pattern and includes a complete traversal that regenerates the source code. This ability to regenerate the code allows easy implementation of analyses that output annotated code, which can then be used as input to further analyses. Additionally, Barat provides flexibility beyond the visitor pattern by supporting association of attributes with AST nodes, thus providing the means for complex analyses which require caching information about a particular node. A Java parser such as Barat would form an integral part of a TIP for Java.

5 Integration via APIs

Deep analyses require sophisticated reasoning facilities. Extensive research into decision procedures, constraint solving, and optimization has produced a library of foundational tools. Many of these implementations are ready for inclusion in a TIP. Some effort should be expended on standardizing APIs and porting APIs to languages such as C++, Java, and O’Caml. We highlight a few examples of foundational tools in this section. Other examples include tools for linear and semidefinite programming and quantifier elimination for polynomials [19]. Mathematical packages like Mathematica [64], which we describe, provide some of these tools.
### Table 4. Summary of foundational tools. Note that BANE is actually a framework for specifying constraint-based analyses for C.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Domain</th>
<th>Language API</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision Procedures</td>
<td>See text</td>
<td>O’Caml, C/C++, Lisp, etc.</td>
<td>Yes</td>
</tr>
<tr>
<td>Chaff</td>
<td>SAT</td>
<td>C/C++</td>
<td>Yes</td>
</tr>
<tr>
<td>PPL</td>
<td>Polyhedra</td>
<td>C/C++</td>
<td>Yes</td>
</tr>
<tr>
<td>BANE</td>
<td>Set constraints</td>
<td>SML/NJ</td>
<td>Yes</td>
</tr>
<tr>
<td>Mathematica</td>
<td>Mathematics</td>
<td>C/C++</td>
<td>No</td>
</tr>
</tbody>
</table>

**Decision Procedures** Ground decision procedures based on Nelson-Oppen or Shostak combination include the ones in STeP [9] and PVS [55] and the standalone procedures SVC [6], CVC [60], CVC Lite [23], and ICS [33]. They offer various logics, but usually include at least propositional, ground linear arithmetic for integers and reals, uninterpreted functions with equality, data structures, arrays, and bit vectors. Several offer APIs. ICS provides APIs for O’Caml, C/C++, and Lisp, while CVC Lite offers a C/C++ API. Decision procedures are already essential components of, for example, hardware verification (e.g., [12]), model verification via verification conditions (e.g., STeP [9], PVS [55]), predicate abstraction (e.g., SLAM [5], BLAST [38]), “infinite bounded” model checking (e.g., SAL [25]), and invariant generation. Undoubtedly, the role of decision procedures will only increase in software analysis.

**SAT Solvers** SAT solvers have improved significantly over the last decade. Chaff [50] is one of the best solvers currently available. Because it is based on the DPLL algorithm, it is complete. However, unlike earlier DPLL-based algorithms, its combination of learning and efficient data structures for Boolean constraint propagation makes it blazingly fast. SAT solvers are already used in many applications for hardware, including model checking, simulation, and fault discovery. Bounded model checking can be applied to software analysis via predicate abstraction. SAT solvers are also an integral part of modern decision procedure packages. Additionally, liveness analysis and reaching definition analysis can easily be encoded into 2SAT, which can be solved in polynomial time, suggesting that SAT may be suitable for higher level analyses, as well.

In a similar vein, BDDs have recently been used for program analyses, specifically, alias analyses [63]. BDDs and SAT solvers will most likely find increasing use as optimized representations of finite sets and relations. The work in [63] includes a tool for manipulating sets at a high level via a BDD implementation of DataLog.

**Polyhedra** The polyhedra domain is one of the most popular abstract domains used in invariant generation in the abstract interpretation framework. Efficient manipulation of polyhedra is essential for these methods to work. The Parma Polyhedral Library (PPL) [4] is an example of a specialized library that manip-
ulates polyhedra. PPL offers several advantages over other polyhedra libraries. First, memory is managed internally so that the client need not provide a static amount of memory in advance. Second, it implements an API for manipulating not necessarily closed (NNC) polyhedra. This feature makes reasoning about strict inequalities clean. The PPL has a C and C++ API, although porting it to other languages is not difficult. For example, the authors have developed a rudimentary interface for Mathematica [64].

**BANE** BANE (Berkeley Analysis Engine) [2] is a framework for writing C constraint-based analyses, such as race detection and points-to analysis. CQual [34], for example, is implemented in BANE. Thus, it differs from the purely foundational tools listed above in that it is a framework, rather than a foundational tool. However, reformulating the solver as an API tool for use with other types of analyses seems like a natural step. Analyses are presented as set constraint generators; BANE solves the generated constraints [1]. Many analyses have been written within the BANE framework. The successor to BANE is Banshee [3].

**Mathematica** Mathematica [64] provides a powerful array of mathematical tools. Additionally, the community of Mathematica users has amassed a remarkable library of tools written in the Mathematica programming language, which includes a subset of ML-like syntax. We have found it invaluable for implementing prototypes of our research.

Mathematica also provides MathLink, an API for interfacing with C/C++ programs. MathLink provides two types of functionality. First, external programs can be integrated within the Mathematica environment, providing, for example, fast implementations of certain functionality. Second, the Mathematica kernel can be called as a tool, making available its vast library of functionality.

### 6 Tool Integration: A Researcher’s Experience

In this section, we provide an account of some of our recent efforts to apply the results of our research to C code, as an example of practical tool integration.

Our first attempt at integration was to apply a termination analysis to C. We took the abstraction approach briefly outlined in Section 2.2. In more detail, we had developed a number-based termination analysis. As with many deep analyses, it is defined in terms of a convenient abstract language, in this case a set of guarded commands. Variables are reals; guards are conjunctions of polynomial inequations; and commands are simultaneous assignments of polynomial expressions to variables. We had already implemented this language of loops and the accompanying analysis in Mathematica [64]. For a prototype, this choice made sense, as the analysis relies heavily on manipulating polynomials.

The challenge, then, was to extract polynomial guarded sets from C source. A sound analysis requires at least the following features:
- **Program slicer.** The slicer extracts a *number abstraction* from a given loop. The remaining variables are number types, while r-values are polynomial expressions (in which no flooring occurs). Guards are replaced by nondeterministic choice where necessary.

- **Control flow graph (CFG).** Even though the analysis examines one loop at a time, the abstracter must verify that functions called within the loop do not modify variables in the number abstraction. Modified variables are removed and the effects of the removal propagated, unless the modification can be represented in the number abstraction.

- **Alias analysis.** First, the alias analysis is necessary for constructing a CFG because of function pointers. Second, the alias analysis is necessary for determining that variables are not modified via aliases.

- **Invariant generation.** Invariants are necessary for specifying initial conditions. Additionally, the subjects of unsigned casts must be determined to be nonnegative at the cast point.

- **Abstracter.** The abstracter finally generates the set of guarded commands, possibly with an assertion of the initial conditions.

As our goal was merely to test the scalability of the technique itself, we implemented only the slicer and abstracter as a CIL analysis. It produces abstractions that are then analyzed by Mathematica. An attempt to communicate between the CIL analysis and Mathematica in an elegant fashion failed. Mathematica’s MathLink provides an interface for C/C++, but CIL is written in O’Caml. We thus used O’Caml’s facilities for interfacing with C, writing a short MathLink interface for O’Caml. Unfortunately, when we reached the compilation stage, we discovered that the MathLink and O’Caml Unix libraries share three function names, stopping linking in its tracks. A second attempt at elegant integration was to write a Mathematica server with which our application could communicate. We have yet to make it work as fast as simply calling `math`. Our final integration matches our initial “hacky” one. Perhaps there is a lesson in integration in this.

This implementation works well enough for timing information, but the analysis is unsound. We initially explored the other components of a sound implementation described above, but postponed the effort.

Classical alias analyses are intended for compiler optimization, not verification and bug finding. We realized that a verification-oriented alias analysis requires a fine level of detail. For a larger C analysis project, which will exploit the termination analysis (thus justifying our postponing the soundness effort), we have written an alias analysis with the following characteristics:

- **Context sensitivity.** Heap allocation points are identified by location and function call context so that they are not unnecessarily conflated. Moreover, heap allocation points have a *unique* attribute, identifying if they are created within the context of a loop or not. If some call in the call context occurs inside a loop, then the allocation point is not unique.

- **Field sensitivity.** Structures play a large role in C programs. Conflating fields is not an option. Essentially, the analysis provides three “types” of locations: *Basic*, for *ints*, *doubles*, *int *s*, *etc.*; *Structure* of locations; and *Array* of
a location, which conflates all positions into one. Such loose typing is almost insensitive to C’s casting features.

- Support for interprocedural dataflow analysis. Our larger analysis framework presents clients with an interprocedural dataflow interface (currently sans recursion). As aliasing is a crucial part of the framework, the client implementation of the transfer function is presented with the current C statement and the alias context.

- Integration with “interprocedural” aspect of interprocedural analysis. The alias analysis naturally provides the possible functions for a function pointer dereference, facilitating the CFG exploration.

While this level of sensitivity has higher overhead than is acceptable for code optimization tasks, it should facilitate verification and bug-finding efforts.

One goal in our current implementation work is to separate the analysis code from the language-specific code. Thus, for example, in the alias analysis, approximately 60% of the code is C-independent: it could be applied to Java just as easily (hopefully). However, other modules in the project do not produce as favorable a statistic. Indeed, we will count ourselves lucky if in the end half our code is independent of C. Of course, as we discussed in Section 3, one potential solution to this problem is to create a common intermediate language to which all (interesting) languages can be compiled.

7 Tool Integration: An Industry Perspective

Results of research in analysis techniques do not find their way into industry, at least not easily. Microsoft is adopting predicate abstraction and model checking to verify drivers; however, it took a research lab to effect the transfer. Even reasonably successful methods developed in industry itself are abandoned by that same industry: ESC/Java, developed at DEC/Compaq research lab, did not survive the acquisition by Hewlett Packard; it has now been taken over by academia. There seems to be a serious gap between research results and industry needs. A single portal with integrated tools may reduce the barrier.

What must such a portal provide? Three ingredients are necessary: (1) A GUI-based platform for invoking analyses and presenting results, (2) a back end that integrates the reports of multiple analyses into a single report, and (3) well-defined API’s that allow new methods to integrate seamlessly with existing functionality, making the ongoing research process as invisible to the user as possible.

At present, there exists one platform that provides such single-point access: Eclipse [29]. It is an ambitious project with the sole apparent goal of offering the best tools for any task involving code, in one integrated development environment (IDE). It is an open-source environment (http://www.eclipse.org) with many hooks to facilitate incorporation of plug-in tools. Tens to hundreds of such plug-ins have been provided, including some static analysis tools. Unfortunately, Eclipse does not provide much beyond single-point access. Plug-ins tend to be loosely coupled, as there is no incentive to integrate the tools. This situation
again places the burden on the user to integrate the methods and their results. Clearly, merely producing Eclipse plug-ins is not the integration answer. Rather, one plug-in should present the results of multiple analyses.

Two research efforts have examined techniques for presenting useful information to users. Kremenek and Engler [44] propose a generic \textit{z-ranking} approach for presenting the output of a single analysis. Essentially, the method is based on the observation that code generally has few bugs; hence, most program locations should satisfy a property, resulting in few error reports. Their approach sorts error reports based on the frequencies of successful and unsuccessful analysis checks. Their experimental results are impressive. Rutar, Almazan, and Foster [58] integrate the error reports of multiple Java checkers to isolate Java classes that probably have many bugs. Ranking at this level allows the user to focus on the buggiest sections of code.

One can imagine a back end integrator that accepts the output of multiple bug finding tools, ranks each tool’s report, and ranks code modules based on probable number of bugs. This back end would be tightly coupled with Eclipse’s interface so that the user can easily explore error reports by rank (e.g., double-clicking on an error report displays the associated code) or by module (e.g., color-coding indicates the presence of likely bugs).

Of course, not all program analyses operate independently of the user. Bandera [20] provides a useful GUI for exploring, annotating, and slicing Java code, among other tasks. Undoubtedly, some tools will require specialized GUI plug-ins for full analysis power; however, all tools should provide basic output for offline analysis.

7.1 An Industry Assessment of Static Analysis Tools

We had the opportunity to evaluate some static analysis tools for a technology-based company that is exploring the use of static analysis tools in its development and testing cycle. We focused on two tools for Java, FindBugs, a shallow method that looks for bug patterns, and ESC/Java, a deep method that applies theorem proving to analyze the code. We describe the results of our evaluations in the context of two different modes of use in the software development process, as different uses impose different requirements on the tools and their level of integration.

\textit{Developer-based use} In developer-based use, individual developers are responsible for selecting and applying tools during their own development and testing of software components. In this case, many of the concerns discussed above apply. We found that FindBugs works well in this environment. Not much effort is required to run the tool, and the bug reports are presented in a user-friendly GUI. It does not, however, integrate with other tools, so if other tools are being used, their analysis results will have to be examined separately.

In contrast, ESC requires a large amount of effort by the developer because of the need for annotations in the code. ESC can be run without annotations, but we found that in this case it produces so much output that it ceases to be
useful. As the software developed by the company under consideration is not safety-critical, this amount of overhead cannot be justified.

*Automated use* Another way to use analysis tools in the development process is to integrate them in the automatic build process: analyses would be triggered by check-in of software modules. No user interaction should be required in this case and reports generated should be ranked in importance, with the option of filtering warnings that are deemed irrelevant at the given stage of development. Neither of the tools considered met these requirements.

The conclusion of our study was that the tools evaluated did not support the needs of this company.

# 8 Conclusion

In this paper we discussed the gap between theory and practice in software analysis and the roadblocks faced in trying to bridge this gap. The paper is complementary to the previous paper prepared for the CUE initiative [7], which discussed the theory and practice of design methodologies.

It is clear that software analysis will become increasingly important. Market pressure will drive developers to produce ever more sophisticated systems that they do not (fully) understand. Regulatory agencies will not be able to stop this infusion of poorly understood software into all aspects of life. It is imperative that we get some handle on these systems. Enforcing sound design methodologies that produce "analyzable code" may work in some restricted environments such as NASA. In general, however, we must expect to be faced with code that is put together under pressure by people with varying levels of skills. To make an impact we must be prepared to apply our methods to the code base produced in this way.

In some sense the software producing world, represented by the body of written software, may be viewed as a natural phenomenon that can be studied like a biological system. Creating the ability to apply analysis methods to this system will create a view that may stimulate entire new research areas, like the invention of the telescope did for astronomy, or the sequencing of the Human Genome for biology. Proper access to the system of study will accelerate research by providing immediate feedback on the effectiveness of methods. In contrast to astronomy and biology, the system under study is an active participant that is itself extremely hungry for metrics that measure performance, reliability, productivity, and complexity. Additionally, it can modify itself through the mere presence of effective analysis methods, resulting — hopefully — in more reliable systems.

As argued in this paper, much theory and many techniques and even prototype tools already exist. The research problem is to apply these methods in an effective way to the software world. It is important that building this bridge to industry is viewed as a legitimate research endeavor, both by funding agencies and by the research community. We believe that the current state of the art
in theoretical software analysis could already provide huge benefits to practical software development, once it is made accessible. It does not, however, stop there. The feedback that we receive via this bridge will accelerate research in foundational methods and tools. The present technology push from the side of formal methods will change into a technology pull from the side of industry. A new decision procedure will be celebrated like advances in linear programming are today. Progress will be self-reinforcing: the research will receive more attention, encouraging more funding and more participation.

The potential benefits are sufficiently large and the needs sufficiently urgent that multiple approaches to this problem should be considered and supported, including tool repositories such as the one initiated by SRI, integrated platforms such as Eclipse, and portals that provide web-based analysis services. The question remains: how do we make it happen?

8.1 Topics for Discussion

Following are some of the potential obstacles that we think need to be addressed:

Implementation Generally, forces in theoretical computer science reward publications, not implementations. How can we encourage more implementation effort?

Considerations:

- We are computer scientists. Is implementation of research work a respectable research pursuit?
- Most researchers have more ideas than time. Is it worthwhile to spend our time on implementation?
- MS and Ph.D degrees require largely individual research contributions.
- Authorship often goes to the developer of the idea; implementors are considered secondary.
- Some implementations, such as CIL, require much effort. Yet, once implemented, they significantly decrease the effort of others. How can the original implementor get appropriate credit for his effort?

Ideas:

- A conference for which submissions consist of paper and implementation, such that the implementation meets certain standards, for example, integration in a preset framework.
- A recognized journal of foundational tool descriptions and documentation, especially for tools that help researchers such as parsers and decision procedures.

Funding structure How can funding agencies be convinced to support implementations of foundational tools and integration efforts?

Considerations
Funding agencies often require “revolutionary new approaches,” explicitly stating in their calls for proposals that extensions to current work will not be considered. This ban immediately excludes any proposal to implement and integrate results of previous research. At best, it encourages ad-hoc implementations that serve merely to demonstrate the feasibility of the methods. As grants usually have durations of up to three years, it is unlikely that significant new methods are developed and implemented within one grant.

Ideas

- Funding agencies could instate special implementation grants that are follow-up grants to the regular grants. These grants could have the requirement that implementations be delivered to some repository. Evaluation of proposals for these grants could take into account the potential of the method and the proposer’s past record of delivering implementations.

Working group for TIP Suppose we agree to instigate a “working group” to focus on creating something like a TIP. What are the challenges?

Considerations

- An important aspect of a TIP is agreement on APIs and other standardization. How do we reach agreement? Who should be involved?
- Should implementation of core integration functionality be an academic pursuit?
- Should implementation of a compiler-like front end be an industry pursuit?
- Is academia suited for this endeavor? How do we ensure continuity? Do we want industry to join in the effort, or just to provide input?
- Who funds the core effort?

Ideas

- Integrate the working group with a tool integration conference.
- More than just providing a TIP, the effort maintains a single website as a portal into the software debugging/verification community. The portal provides the integrator and supporting tools for download, a list of publications (refereed and accepted tools), and a submission page for new tools.
- The refereed aspect of tool submission provides a source of prestige, which in turn would increase the number of submissions. Moreover, it would enforce robustness, high quality of output, and adherence to agreed upon standards. Such high quality software would in turn promote its use, and then the prestige of getting a tool accepted.

Enforcement or voluntary adoption Should analysis techniques be forced upon industry by government intervention? Or should we have to attract industry?

Considerations
At present, agencies like the FAA and the FDA impose mostly process-based standards rather than product-based requirements. With increasing system complexity, will process-based standards be sufficient to guarantee public safety? Could agencies such as the FAA and the FDA, and perhaps NIH (and similar agencies in Europe), take a more active role in promoting the development and implementation of analysis methods?

Government intervention could put too many requirements on the research community.

Many applications need not be safe. Application of analysis methods for these applications is mostly governed by economic concerns, such as time-to-market and competitive advantage of higher product reliability. In this case, application of analysis methods must be demonstrated to be justifiable.

Usability is an issue either way.

What is the role of product liability? Do companies use the lack of efficient analysis methods as an excuse to evade product liability? Or could the threat of product liability promote the use of analysis methods?

References

Aspects-oriented hard real-time programming and tool integration

Wolfgang Pree and Josef Templ
Department of Computer Science, University of Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Abstract
The paper sketches how aspects, integrated into a hard real-time programming language, could improve the modularization of embedded software. Based on an overview of the Timing Definition Language (TDL, as described in the TDL language report [1]), we present how aspect-oriented concepts could be woven into that language. A discussion of the advantages and disadvantages of AspectTDL rounds out the paper.

1 Motivation
Deterministic hard real-time programming systems such as Giotto [2] or TDL [1] assume to interoperate with their environment through the usage of so-called sensors and actuators exclusively. A sensor represents a value provided by the environment. It is read into the real-time system by using a getter function, i.e., a function that returns the value of the sensor by accessing appropriate hardware registers or other environment variables. An actuator represents a value to be set in the physical environment by means of a setter procedure that writes the value to an appropriate hardware register or other environment variable. Getters and setters are the only means to communicate with the physical environment of a real-time system and they are implemented outside of the source code of the real-time system, for example, in C or Java.

Getters and setters are called from the real-time system by means of internally maintained glue code, called drivers. The execution system of real-time code simply calls the appropriate driver in order to activate a getter or setter. The driver concept abstracts from the concrete implementation of getters and setters.

Now suppose that we want to equip a real-time system with an additional feature called limit monitoring. This allows a user to select a subset of sensors and actuators and to specify upper and lower bounds for the values and associated actions in case of leaving the specified range. Such a feature would affect all getters and setters insofar as code for range checking must be included for all sensors and actuators that were selected for monitoring by the user. Clearly this is an undesirable global change of program code spread across many different source files. In other words this would be a so-called crosscutting aspect, which is the subject of aspect oriented programming (AOP) as pioneered by G. Kiczales et al. [3]. Another example of a crosscutting aspect is integration, which means that the real-time system is integrated into some non real-time environment used to visualize the process state and to interact with the user. The interaction between heterogeneous real-time systems in a distributed system may also be seen as a crosscutting aspect.

Without going into the details of AOP implementation, it should be mentioned here that the getter and setter functions are a poor place for implementing crosscutting aspects of a real-time system. The drivers which are used to abstract from the getter and setter implementation are much better place. If a driver provides the means for incorporating additional functionality, aspects may be incorporated even dynamically without the need to change existing code.

The following sections describe the concepts of AOP, the relevant features of the hard real-time language TDL and refine the ideas of AOP in the context of real-time software.

2 Multi-dimensional modularization through aspects
Aspect-oriented programming (AOP, [3, 4, 5]) was pioneered by Gregor Kiczales mid of the 1990s as a means of overcoming the problems related to one static modularization as prevalent in module- and object-oriented programming languages. The premise of AOP is that there is not one perfect static modularization, but that various different modularizations are required to modularize a software system. This is why we view AOP as a means of multidimensional modularization. The additional modularizations that override the one static modularization depend on conditions that are evaluated at run-time.

Conceptually, AOP is based on the more general idea of meta-programming, which had already been refined by Gregor Kiczales. Meta-programming allows the modification and extension of the semantics of a
programming language by writing code that defines the specific semantics. For example, a method invocation in an object-oriented language such as Java can be modified so that additional processing is done, either before or after the actual method invocation. AOP applies meta-programming for improving the modularization of a software system. In the following we summarize those AOP concepts and terms that are relevant in the context of this paper.

Figure 1a schematically shows the static modularization of a software system. Each bar represents a module. The length of a bar corresponds to the number of lines of code. Those parts of the source code that would logically form a module but which had to be scattered over several modules are shown as gray bars in the modules. Source code that logically belongs together but is split over several modules has the disadvantage that changes have to be accomplished in several modules, not in one spot. Thus, it becomes difficult to maintain consistency. The AOP advocates argue that it is impossible to find one perfect modularization for a real-world software system that does not encounter the sketched problem. If one changes the static modularization to solve one detected modularization glitch, the modularization of other aspects that were originally well modularized, might be distorted. In other words, according to AOP one static modularization is not sufficient.

Figure 1. AOP cross-cutting

In order to solve the problem of scattered code that logically forms a unit, AOP allows us to cross cut several modules, that is, to cut the scattered code and put it into an additional module. The gray rectangle in Figure 1b represents such an additional module containing the code that was originally scattered over several modules. After presenting the Timing Definition Language we show how aspects could be integrated into that language.

3 The Timing Definition Language (TDL)

Giotto [2] with its sound formal semantics provides language constructs that allow the development of deterministic, composable control applications. Their timing and communication behavior can be specified independent of their implementation. The Timing Description Language (TDL, [1]) enhances Giotto towards a component architecture for real-time control applications. Besides some syntactical changes and minor adaptations of the semantics, however, TDL is based on the same programming model as Giotto.

Figure 2. Visual Representation of a TDL module

Giotto programs are multi-mode and multi-rate systems that periodically execute a set of tasks and other activities. Figure 2 shows a simplified, visual representation of a Giotto program, which corresponds to a TDL
A TDL application comprises one or more modules. A module consists of a set of modes. A mode contains a set of activities: task invocations, actuator updates, and mode switches. A TDL module is in one mode at a time. A mode defines the set of tasks that needs to be executed in parallel in a particular mode of operation of the real-time application. In addition to executing tasks, a mode also defines actuator update operations and the conditions for mode switches. All kinds of activities (task invocation, actuator updates, and mode switches) can be activated with different rates. In order to provide deterministic behavior of a control system, Giotto defines that task outputs are only available at the end of a task’s logical execution period, which is also called the FLET (fixed logical execution time) assumption. The results of a task may be available earlier in internal program variables, but each result value that is visible to clients of a task is updated exactly at the end of the task’s FLET.

Figure 3. The Giotto/TDL Task Model

FLET provides the cornerstone to deterministic behavior, platform abstraction, transparent distribution, and well-defined interaction semantics between different activities. It is always determined which value is in use at which point in time and there are no race conditions or priority inversions involved. Figure 3 shows the FLET property of Giotto/TDL tasks.

In addition to the drivers mentioned in the motivation section, TDL uses drivers for all task events and for mode switches, where the latter are not shown above.

- the release driver is called at the beginning of the task’s FLET by the so-called E-machine, part of TDL’s run-time system,
- the start driver is called by the so-called S-machine, also part of TDL’s run-time system, in order to invoke the task’s implementation function,
- the stop driver is called by the S-machine after finishing execution of a task and
- the termination driver is called at the end of the task’s FLET by the E-machine.

It is beyond the scope of this paper to describe the details of the driver implementations, however, it should be clear that the drivers are the key of an AOP implementation for TDL. They allow for extending the semantics of various runtime activities without the need to change the E-machine or the functionality code.

Differences between Giotto and TDL

The most visible syntactical differences between TDL and Giotto are:

- the introduction of a top level language construct (module) and the reorganization of mode declarations, where ‘start’ is a modifier of a mode declaration in TDL
- the elimination of global output ports, which are replaced by task output ports in TDL,
- the elimination of explicit task and mode drivers, which are merged into mode declarations in TDL,
- the addition of constants, which may also be used to initialize ports in TDL,
- the introduction of units for timing values in TDL.

The following list explains differences between TDL and Giotto semantics.

- **program start.** A TDL program is started by switching to the start mode. This means that at time zero, there are neither actuator updates nor mode switches. In Giotto, the actuator updates and mode switches of the start mode take place at time zero. There are, however, no further actuator updates or mode switches of the target mode at time zero.
- **non-harmonic mode switch.** Giotto allows to switch a mode even if there are running tasks as long as those tasks exist with the same task period in the target mode. However, there may be delays involved when switching to the target mode. Furthermore, the task will deliver output values to the target mode, which do not correspond to inputs specified there. TDL does not allow non-harmonic mode switches. We
are thinking about alternative ways of performing even faster mode switches without the need to continue running tasks in the target mode, with simpler semantics and, last but not least, without any delays.

- **Deterministic mode switch.** Giotto requests that among all mode switch guards of a mode only one may return true at a particular point of time. In contrast, TDL evaluates mode switch guards in textual order from top to bottom and performs the first mode switch whose guard returns true. This definition allows a more efficient implementation and preserves determinism.

- **Actuator update.** A guarded actuator update in Giotto means that the actuator setter is called independently of the guard’s result. In TDL, actuator update and actuator setter are both guarded and performed only if the guard returns true.

The following list describes tool related differences between TDL and Giotto.

- **E-code file format.** TDL defines a binary, platform independent E-code file format and uses statically typed APIs for connecting programs with external functionality code. Platform specific output may be generated by a platform plugin for the TDL compiler.

- **E-code instructions.** The structure and semantics of Giotto E-code instructions has not been changed in TDL but a SWITCH instruction has been added. It is used to perform mode switches. In Giotto, mode switches are performed by the JUMP instruction by jumping to code of a different mode. The SWITCH instruction makes this special usage of JUMP explicit and thereby simplifies the detection of mode switches in the E-machine.

- **Time Resolution.** TDL uses microseconds internally for all timing values, whereas Giotto is based on milliseconds. This means, that TDL programs may use mode periods below 1 millisecond, given that the underlying E-machine is fast enough.

**4 AspectTDL**

The following example illustrates our ideas regarding an aspect-oriented extension of TDL. Note that this is an experimental syntax and that we have not yet implemented the aspect-oriented extensions of TDL. The presentation should lead to discussions regarding, for example, the pros and cons of aspects in the context of TDL or which level of genericity such an extension should provide. The syntax will significantly be shaped by the implementation of these language features.

The basic concept of AspectTDL is that the various entities described in TDL, associated with various TDL modules, are subject to cross-cutting. For example, the developer could define an extra module as aspect that deals with a subset of tasks that are defined in the already specified modules.

We use the TDL module construct, which is the primary unit of program decomposition in TDL, as a container for aspect-oriented declarations. Following the terminology introduced for aspect-oriented Java [4], we declare aspects by using the keyword *aspect*. A module may declare any number of aspects.

```java
module SampleAspectTDLModule {
    aspect limo { //assume: limit monitoring on task outputs
        pointcut limoTasks (task): uses limoPointcutImpl;
        after terminate [wcet=1us] uses limoImpl;
    }
    aspect connect { //assume: output und actuator ports will be connected to the environment
        pointcut connectedPorts (output || actuator): uses connectPointcutImpl;
        before set [wcet=2us] uses connectImpl;
    }
    aspect watchdog { //assume: mode switches will be monitored
        pointcut watchSwitches (mode): uses watchPointcutImpl;
        after switch uses watchImpl;
    }
}
```

An aspect has a name, which serves to describe the basic idea associated with the aspect. The details of the aspect are defined within the aspect's body, which is surrounded by a pair of curly brackets.
Every aspect needs a definition of TDL program entities that are the subject of the aspect. We use the keyword `pointcut` for defining the selection of those subjects. In addition, an advice (before or after) must be specified which will be executed whenever the selected program entity performs a specific action.

A pointcut always refers to one of the various TDL program entities such as task, sensor, actuator, output or mode. The filter specified in the parentheses may be an elaborate expression or simply an appropriate keyword. In addition to selecting the program entities in the aspect declaration the user may specify a filter using an external function (e.g., `limoPointcutImpl`). This function must be implemented as external functionality code.

An advice is either performed before or after the selected program entity performs a specific action, which depends on the kind of entity. A task, for example, may be released, started, stopped, and terminated. An output or actuator port may be set or a mode may be switched to. An advice may need a non-negligible amount of CPU time, which can be specified by an optional wcet-annotation (worst case execution time). In any case the implementation of an advice will be done in external functionality code.

**Implementation strategy**

The basic idea of implementing aspects in TDL is to augment the driver architecture of the TDL E-machine. Every action performed on a program entity is invoked by using a so-called driver, which serves as the glue code between the E-machine and the external functionality code. By applying the Observer pattern [6] to the notion of a driver, we arrive at a framework that allows the registering of listeners for driver events. In addition to executing a driver's code, we prepare the hooks needed for pre- and postprocessing as sketched in the following pseudo code:

```java
void terminateDriver() {
    for each b in beforeTerminateListeners: b.call();
    driver code;
    for each a in afterTerminateListeners: a.call();
}
```

A listener object is created for each task that belongs to a particular aspect. The developer has to provide the corresponding pointcut implementation function which would look like the following pseudo code:

```java
boolean limoPointcastImpl(Task t) {
    return (aspect limo applies to t);
}
```

Returning true means that a listener is to be created for the particular task and the listener has to be registered. Registering a listener means that in case of a before advice the listener is added to the corresponding 'before...' list and in case of an after advice the listener is added to the corresponding 'after...' list.

**5 Conclusions**

From the evidences outlined in the paper we believe that weaving aspects into a hard real-time programming language such as TDL would help to improve the modularization of embedded software. As mentioned in the motivation, it also facilitates the integration with legacy systems or with non-real-time environments.

We have integrated TDL with the Simulink tool chain [7] so that it becomes useful for our cooperating industry partners. In this context, the aspect-oriented extension was inspired as we had to think of requests such as the one how limit monitoring could be specified. The aspect-oriented extension of TDL seems to be the adequate means for that and for analogous future requirements. In other words, an AspectTDL will significantly improve the already accomplished integration of TDL within the Simulink environment.

A potential disadvantage is the additional complexity of TDL. Furthermore, the processing of aspects leads to a run-time overhead that interferes with the Giotto/TDL paradigm of synchronous input and output.

A first implementation of AspectTDL might imply changes to the syntax of AspectTDL. Several details, such as the syntax of filters, still need to be defined.
Bibliography


Abstract

We present a technique that enables the focused application of multiple analyses to different modules in the same program. In our approach, each module encapsulates one or more data structures and uses membership in abstract sets to characterize how objects participate in data structures. Each analysis verifies that the implementation of the module 1) preserves important internal data structure consistency properties and 2) correctly implements an interface that uses formulas in a set algebra to characterize the effects of operations on the encapsulated data structures. Collectively, the analyses use the set algebra to 1) characterize how objects participate in multiple data structures and to 2) enable the inter-analysis communication required to verify properties that depend on multiple modules analyzed by different analyses.

We have implemented our system and deployed three pluggable analyses into it: a flag analysis for modules in which abstract set membership is determined by a flag field in each object, a plugin for modules that encapsulate linked data structures such as lists and trees, and an array plugin in which abstract set membership is determined by membership in an array. Our experimental results indicate that our approach makes it possible to effectively combine multiple analyses to verify properties that involve objects shared by multiple modules analyzed by different analyses.

1 Introduction

Data structure consistency is important for successful program execution — if an error corrupts a program's data structures, the program can quickly exhibit unacceptable behavior or even crash. Motivated in part by the importance of this problem, researchers have developed algorithms for verifying that programs preserve important consistency properties [3, 7, 13, 29–31]. However, two problems complicate the successful application of these kinds of analyses to practical programs: scalability and diversity. Because data structure consistency often involves quite detailed object referencing properties, many analyses fail to scale. Because of the vast diversity of data structures, each with its own specific consistency properties, it is difficult to imagine that any one algorithm will be able to successfully analyze all of the data structure manipulation code that may be present in a sizable program.

This paper presents a new perspective on the data structure consistency problem. Instead of attempting to develop a new algorithm that can analyze some specific set of consistency properties, we instead propose a technique that developers can use to apply multiple pluggable analyses to the same program, with each analysis applied to the modules for which it is appropriate. The analyses use a common abstraction based on sets of objects to communicate their analysis results. Our approach therefore enables the verification of properties that involve multiple objects shared by multiple modules analyzed by different analyses.

1.1 Target Application Class

Our technique is designed to support programs that encapsulate the implementations of complex data structures in instantiatable leaf modules, with these modules analyzed once by very precise, potentially expensive analyses (such as shape analyses or even analyses that generate verification conditions that must be manually discharged using a theorem prover or proof checker). The rest of the program uses these modules but does not directly manipulate the encapsulated data structures. These modules can then be analyzed by more efficient analyses that operate primarily at the level of the common set abstraction.

We have implemented our analysis framework and populated this framework with three analysis plugins: 1) the flags plugin, which is designed to analyze modules that use a flag field to indicate the typestate of the ob-
objects that they manipulate; 2) the PALE plugin, which implements a shape analysis for linked data structures (we integrated Anders Møller’s implementation [30] of the Pointer Analysis Logic Engine analysis tool into our system); and 3) the array verification plugin, which generates verification conditions for consistency properties of array-based data structures. Verification conditions from the array verification plugin are designed to be discharged manually using the Isabelle interactive theorem prover. We have used our analysis framework to analyze several programs; our experience shows that it can effectively 1) verify the consistency of data structures encapsulated within a single module and 2) combine analysis results from different analysis plugins to verify properties involving objects shared by multiple modules analyzed by different analyses.

1.2 Contributions

The contributions of this paper are the following:

- **Pluggable Analysis Framework:** We show how to apply multiple analyses to multiple data structures encapsulated within multiple modules, with the analysis results appropriately combined to verify properties that span multiple modules. The approach supports sharing patterns in which objects move between different data structures and patterns in which objects participate in multiple data structures simultaneously.

We introduce abstract sets as the key abstraction that each analysis uses to characterize how objects participate in encapsulated data structures. The connection between sets and concrete data structure consistency properties enables modules to express the data structure participation requirements that externally accessible objects must satisfy without exposing the data structure representation to their clients. The set abstraction also enables different analyses to interoperate to verify properties that span multiple data structures and modules.

We show how to use the common set abstraction to specify and verify global invariants that correlate membership of objects in different data structures analyzed by different analysis plugins (Section 3).

We provide mechanisms — scopes and defaults — which allow developers to write strictly local specifications of procedures, without having to explicitly include global invariants. Our system then automatically conjoins these global invariants when appropriate.

- **Analysis Plugins:** We present three analysis plugins that show how our approach works in practice:

  a flag typestate analysis for modules in which set membership is determined by the value of a flag field in each object (Section 5), the PALE analysis plugin for modules that manipulate linked data structures such as lists and trees (Section 6), and the array analysis plugin that can verify arbitrarily complex properties of array-based data structures by generating verification conditions and discharging them using an interactive theorem prover (Section 7). The typestate plugin can be thought of as a scalable plugin that propagates and verifies membership of objects in global sets; the PALE plugin is an example of a more precise shape analysis plugin; and the array analysis plugin is an extreme point that in principle has no bound on the complexity of properties that it can verify. More precise analysis plugins may require more analysis time or more interaction with the user; this cost is amortized because these analyses are typically applied to instantiatable modules that encapsulate reusable data structures.

- **Experimental Evaluation:** We present our experience using our implemented system to analyze programs that require the use of multiple analysis plugins to verify important consistency and typestate properties (Section 8).

2 Example

We next discuss an example program that shows how to use our approach to verify 1) the consistency of individual data structures encapsulated in instantiatable modules, 2) that the rest of the program uses each module correctly, and 3) important properties that involve data structures encapsulated in different modules. Our example program implements the popular minesweeper game.1 Figures 1, 2, and 3 present a linked list module used in our example minesweeper program. This module has a specification section (Figure 1), an implementation section (Figure 2), and an abstraction section (Figure 3). The abstraction section specifies the relationship between the concrete data structure implementation and the abstract set specification, and enables the PALE plugin to check that the implementation satisfies its specification.

The abstract Content set in Figure 1 represents the contents of the list. (The notation Content' denotes the new version of Content after a procedure executes; the unprimed Content denotes the old version before it executes). The procedures in the List module use this

1Full source code for the minesweeper example and other case studies, the interpreter for our language, and analysis engine is available at http://cag.csail.mit.edu/~plam/mpa.
set to express their preconditions, postconditions, and effects. The requires clause of the add procedure, for example, requires that the parameter e (which the add procedure will insert into the list) not already be in the Content set. The ensures clause states that the effect of the add procedure is to add the parameter e to the Content set. The modifies clause indicates that the procedure modifies the Content set only.

Procedure specifications can also express cardinality constraints. The requires clause of the removeFirst procedure, for example, uses the formula \( \text{card(Content)} \geq 1 \) to require that the Content set be nonempty upon entry.

An analysis based on monadic second-order logic over trees (as implemented in the PALE analysis tool [30]) is able to verify that the List implementation correctly implements its specification. However, it needs some additional information to do so. The abstraction section in Figure 3 provides this information.

This abstraction section starts by identifying the analysis plugin used to verify this module; in this case the PALE analysis plugin. This analysis plugin implements a decision procedure for the monadic second-order logic over trees and uses this decision procedure to analyze procedures that manipulate recursive linked data structures such as lists and trees [20]. To enable the application of this analysis to the List module, the abstraction section identifies the correspondence between the abstract sets in the specification and the concrete data structure encapsulated inside the module. The statement Content = \{x : Entry \mid root<next*>x\}; defines the Content set to be all objects x reachable by following next fields starting from the root variable.

The analysis uses this correspondence to translate the requires, ensures, and modifies clauses (expressed in terms of abstract sets) into properties of the concrete data structure (which in this case are expressed in monadic second-order logic over the objects and fields in the concrete heap). For example, the translated precondition of add is !root<next*>e, which states that e is not reachable by following next fields starting at the root. The analysis then uses the translated requires clause as a precondition and the translated ensures clause as a postcondition of each procedure. Note that other modules need not be aware of how membership in Content is determined; they simply use the Content set in their own specifications, as needed.

So far, we have presented a generic List mod-

---

\footnote{This implementation places the next field directly in the Entry objects. Our approach also supports the more common implementation that uses auxiliary encapsulated list objects to refer to the Entry objects; in that implementation the auxiliary list objects (and not the Entry objects) contain the next fields.}
ule. In the minesweeper example, we instantiate this List module as an UnexposedList, which stores minesweeper board cells which have not yet been cleared. The instantiation mechanism of our language substitutes the generic Entry format with the Cell format used in the rest of the minesweeper example. Conceptually, instantiation creates a fresh copy of the instantiated module, carrying out substitutions as appropriate to make the generic implementation applicable to the particular use at hand.

2.1 Verifying Cross-Module Properties

We next illustrate how our approach enables the verification of properties that span multiple modules. Our minesweeper implementation has several modules (see Figure 4): a game board module (which represents the game state), a controller module (which responds to user input), a view module (which produces the game’s output), an exposed cell module (which stores the exposed cells in an array), and an unexposed cell module (which stores the unexposed cells in an instantiated linked list). There are 750 non-blank lines of implementation code in the 6 implementation sections of minesweeper, and 236 non-blank lines in its specification and abstraction sections.

Our minesweeper implementation uses the standard model-view-controller (MVC) design pattern [15]. The board module (which stores an array of Cell objects) implements the model part of the MVC pattern. Each Cell object may be mined, exposed or marked. The board module represents this state information by contributing isMined, isExposed and isMarked flags to Cell objects. At an abstract level, the sets MarkedCells, MinedCells, ExposedCells, UnexposedCells, and U (for Universe) represent sets of cells with various properties; the U set contains all cells known to the board. The board also uses a flag gameOver, which it sets to true when the game ends.

- Unless the game is over, the set of mined cells is disjoint from the set of exposed cells.
- The sets of exposed and unexposed cells are disjoint.
- The set of unexposed cells maintained in the board module is identical to the set of unexposed cells maintained in the UnexposedList list.
- The set of exposed cells maintained in the board module is identical to the set of exposed cells maintained in the ExposedSet array.
- At the end of the game, all cells are revealed; i.e. the set of unexposed cells is empty.

We next explain how our system verifies the fourth and fifth properties listed above. Note that the board module, which is analyzed by the flag plugin, defines its sets using flag values, so that board’s set of exposed cells consists of all objects with the field isExposed set to true, whereas the ExposedSet module defines its contents set by array membership.

Although our system focuses on using sets to model program state, not every module needs to define its own abstract sets. Indeed, certain modules may not define any abstract sets of their own, but instead coordinate the activity of other modules to accomplish tasks. The view and controller modules are examples of such modules. The view module has no state at all; it queries the board for the current game state and calls the system graphics libraries to display the state.

Because these modules coordinate the actions of other modules — and do not encapsulate any data structures of their own — the analysis of these modules must operate solely at the level of abstract sets. We therefore analyze these modules using a subset of the flag plugin. This subset tracks abstract set membership, solves formulas in the boolean algebra of sets, and incorporates the effects of invoked procedures as it analyzes each module. It does not, however, need to reason about the correspondence between the concrete data structure representations and the abstract sets.

The analysis of the view and controller modules illustrates a core idea behind our approach: we use faster, less-detailed analyses on high-level modules that primarily coordinate the actions of other modules, and apply more-precise analyses to verify leaf modules that encapsulate implementations of sophisticated data structures.

Note that the set abstraction supports typestate-style reasoning at a per-object level (for example, all objects in the ExposedCells set can be viewed as having a conceptual typestate Exposed). Our system also supports the notion of global typestate: for instance,
the board module has a global gameOver variable which indicates whether or not the game is over. Using this variable and the definitions of sets, we maintain the global invariant

gameOver | disjoint(MinedCells, ExposedCells).

This global invariant connects a global typestate property — is the game over? — with a object-based typestate property evaluated on objects in the program — there are no mined cells that are also exposed. Our analysis plugins verify these global invariants by conjoining them to the preconditions and postconditions of methods. Note that global invariants must be true in the initial state of the program; if some initializer must execute to establish an invariant, then the invariant can be guarded by a global typestate property.

Another invariant concerns the correspondence between the ExposedCells and UnexposedCells sets with the ExposedSet.Content and UnexposedList.Content sets.

\[(\text{ExposedCells} = \text{ExposedSet}.\text{Content}) \& \(\text{UnexposedCells} = \text{UnexposedList}.\text{Content}\)\]

Recall that, in our example, the ExposedSet, the UnexposedList, and the board are all implemented in different modules, and are analyzed by different analysis plugins. This invariant verifies that a set defined by field values is equal to a set defined by reachability in the heap, and that a set defined by field values is equal to a set defined by membership in an array.

Our analysis ensures this property by conjoining it to ensures and requires clauses of appropriate procedures. The board module is responsible for maintaining this invariant. However, the flag analysis used for the board module does not, in isolation, have the ability to verify the invariant, because it cannot reason about the heap structure of the program. Because we have a common set specification language, though, the flag analysis can successfully use the ensures clause of its callees, along with its own analysis tracking ExposedCells membership, to guarantee the invariant.

3 Modular Analysis Framework

We next discuss the basic strategy that we expect analysis plugins to implement, and discuss the tasks that they must perform to verify that each implementation section correctly implements its specification. In general, an analysis plugin must ensure that the implementation of a module conforms to its specification, and that any calls originating in the module it is analyzing satisfy their preconditions.

3.1 Implementation Language

Implementation sections for modules in our system are written in a standard memory-safe imperative language supporting arrays and the dynamic allocation of objects. Analysis plugins use our system’s core libraries to easily manipulate abstract syntax trees for this imperative language. Using these libraries, we have implemented an interpreter for our language; it would also be straightforward to write a compiler for our language.

We point out one special feature of our imperative language, which we call formats. Formats aid modular reasoning about shared objects by encapsulating fields while allowing modules to share objects. When the program creates an object with format $T$, the newly-created object contains the fields contributed to format $T$ by all modules in the program [8]. A simple type checker for the implementation language statically ensures that each module accesses only fields that it has contributed to an object. Note that no analysis plugin needs the full layout of an object; it will only need the fields which the module under analysis has contributed to that object.

The implementation language supports (but does not require) assertions and loop invariants, which enable fine-grained communication with the analysis plugin. The syntax of assertions is specific to the analysis plugin used to analyze the module. Assertions are ignored by the implementation language interpreter; once statically verified, they do not affect the run-time behavior of the program.

3.2 Specification Language

Figure 5 presents the syntax for the module specification language. A specification section contains a list of set definitions and procedure specifications, and lists the names of formats used in set definitions and procedure specifications. Set declarations identify the module’s abstract sets, while boolean variable declarations identify the module’s abstract boolean variables. Each procedure specification contains a requires, modifies, and ensures clause. The modifies clause identifies sets whose elements may change as a result of executing the procedure. The requires clause identifies the precondition that the procedure requires to execute correctly; the ensures clause identifies the postcondition that the procedure ensures when called in program states that satisfy the requires condition. Both requires and ensures clauses use arbitrary first-order formulas $B$ in the language of boolean algebras extended with

\[\text{A formal context-free grammar for our language can be downloaded from our publicly-readable Subversion source code repository at http://plam.csail.mit.edu/svn/repos/trunk/module-language/formatlanguage.sablecc.}\]
We have implemented three plugins in our analysis framework: a flags plugin, which assigns set membership based on field values (Section 5), a PALE plugin, which assigns set membership based on heap reachability (Section 6) and an array plugin, which assigns set membership based on array membership (Section 7).

4 Scopes and Defaults

In this section, we present the notions of scopes and defaults. These notions enable developers to write more-concise specifications when using our modular analysis framework.

Scopes serve two purposes: they enable the specification and verification of cross-module invariants by identifying the subset of a program in which an invariant is expected to hold, and they combat specification aggregation by hiding irrelevant sets from callers. Scopes are key to our system’s verification of invariants containing sets from different modules: by designating certain modules as public access points, we ensure that scope invariants always hold outside their declaring scope. Scopes also shield callers from irrelevant detail: only sets from exported modules are visible to modules in different scopes. This serves to bound the detail required in procedure specifications: the specification of procedure $p$ belonging to scope $C$ need only contain the effects of procedures on sets in $C$ and exported sets outside $C$.

Defaults allow procedure specifications to simplify specifications in a different way. Using defaults, the developer can factor out common conjuncts that repeatedly appear in a module’s procedure specifications. These conjuncts need only be written once per module, and are automatically conjoined to procedure specifications for that module, unless they are specifically suspended at a procedure.
we can reason about scopes individually, since they are into scopes along orthogonal axes. For most purposes, independent of each other. We discuss multiple scopes can participate in multiple scopes at the same time; this in the context of calling restrictions. Each module com-

\[ S ::= \text{scope } s \]
\[
\{ \text{modules } M^*; \text{ exports } M^*; \mid \text{invariant } B; \} \}
\]

Figure 7: Grammar for Scope Declarations

4.1 Scopes for Specifying Invariants that Cross Module Boundaries

Consider module Main which calls module Worker. The Worker module uses two helper modules, Inbox and Outbox, which define sets Input and Output respectively. The Worker module itself defines the Jobs set, which satisfies the cross-cutting invariant

\[ I : \text{Jobs} = \text{Inbox.Input} \cup \text{Outbox.Output}, \]

which must hold on entry to Worker and is always ensured upon exit from Worker. Ordinarily, specifications for procedures of Main must therefore include the invariant I in their own preconditions and postconditions to be able to call Worker; worse yet, any transitive caller of Main also needs to include I. We call this problem specification aggregation, and we describe scopes, our solution to the specification aggregation problem.

Syntax of Scopes. Figure 7 presents the syntax of scope declarations. A scope declaration contains a set of modules; a subset of these modules are declared as exported modules. Scope declarations may also contain a scope invariant.

We describe the components of a scope declaration using a typical scope C. Exported modules are accessible from outside C: that is, only procedures in exported modules may be called from outside C, and only sets declared in exported modules may appear in specifications outside C. Private modules belong to C but are not exported. Sets belonging to private modules are private sets. An invariant B is a boolean algebra formula which is guaranteed to be true in the initial state of the program, assumed to hold at all incoming boundary points, and verified at all outgoing boundary points.

Multiple Orthogonal Scopes. Note that a module can participate in multiple scopes at the same time; this multiple participation enables modules to be grouped into scopes along orthogonal axes. For most purposes, we can reason about scopes individually, since they are independent of each other. We discuss multiple scopes in the context of calling restrictions. Each module combines the calling restrictions from all of its scopes: if M is a private module in some scope C, only modules that are also in C can invoke M, and if M is exported in C⁴, only modules that are not in C can call M.

We need only disallow calls to exported modules of C if C has an invariant.

Scope Calling Condition. Our analysis verifies that the program satisfies the following scope calling condition. This condition ensures that the program’s scope invariants hold at scope boundary points, defined below. Let scopes(M) denote the set of scopes C such that C declares M in its modules clause, and let exportingScopes(M) denote the set of scopes C such that C declares M in its exports clause. Let the “private yard” of module M be yard(M) = scopes(M) \ exportingScopes(M). A procedure call from M' to M is allowed if and only if M is exported in precisely the scopes C ∈ scopes(M) \ scopes(M') of the scope difference. More precisely, we say that module M' calls module M if the body of some procedure in the implementation of module M' contains a call to a procedure declared in module M. We then require the following inter-scope call condition to be satisfied for every pair of modules (M', M): if module M' calls module M, then

\[
\text{scopes}(M) \setminus \text{scopes}(M') \subseteq \text{exportingScopes}(M) \]
\[
\land \text{scopes}(M) \cap \text{scopes}(M') \subseteq \text{yard}(M).
\]

The first conjunct of the calling condition ensures that the incoming boundary points are the only points at which execution can enter a scope, and the second conjunct ensures that between any two instances of incoming boundary points in an execution trace, at least one outgoing boundary point occurs.

Semantics of Scopes and Invariants When our analysis successfully verifies a program, it is certifying that each scope invariant holds in the defining scope’s exterior: boundary points separate the interior of a scope from its exterior. Inside a scope, the invariant may be temporarily violated; our analysis then checks that the invariant is restored before the program exits the scope. The semantics for invariants is therefore that the scope invariant may be assumed to hold upon entry to its scope, and the invariant must be verified that the scope invariant holds upon exit.

The set of incoming boundary points is defined as the set of the entry points of procedures for exported modules of C and return points for potentially-reentrant call sites inside C. (A return point for a call site is the immediate control-flow successor of the return statement in the call site’s target; potentially-reentrant call sites are those that directly invoke a method outside C which, on some execution trace, transitively call back into C.) The outgoing boundary points are defined as the exit points of exported procedures, plus all potentially-reentrant sites calling outside C belonging to procedures inside C. Our analysis also checks that from inside C, incoming boundary points will not be
called; in the interior, only procedures belonging to private modules of \( C \), or outside \( C \) entirely, may be called.

Our system enables the verification of invariants by placing two constraints on how invariants are defined. First, all invariants must be true in the initial state of the program\(^5\). Second, an invariant in scope \( C \) may only refer to sets and booleans in scope \( C \).

**Soundness of Assuming Invariants.** We justify our handling of invariants by arguing that whenever we assume an invariant, it must already be true in the underlying dynamic program state. Since the only possible unsoundness in our handling of invariants comes from assuming the invariant, we can show soundness simply by proving that an invariant is always true when our treatment assumes that invariant. In general, invariants will not be directly provable at calling sites by our analysis, because the sets mentioned in the invariant may be private sets invisible to the caller, implying that no information is available about these sets outside the scope, and in particular at the calling site.

Our condition on calling incoming boundary points from within a scope gives us the following property:

**Proposition 1** *(Boundary Point Nesting)* For all scopes \( C \), all execution traces \( s_0, \ldots, s_n \) and all pairs \( s_p, s_{p+1} \) of incoming boundary points, there exists an outgoing boundary point \( s_q \) such that \( p < q < p_{i+1} \). Similarly, between any two outgoing boundary points is an incoming boundary point.

Two incoming boundary points will never be adjacent, because our analysis verifies that only procedures belonging to private modules of \( C \) or the exterior will be invoked inside \( C \); in the first case, there is no incoming boundary point, and in the second case, we have constructed the outgoing boundary points such that there will always be an outgoing boundary point when the call is potentially-reentrant. Two outgoing boundary points will never be adjacent: after an exit from an exported procedure, control must flow to a procedure outside \( C \) (since only the exterior can call an exported procedure), and after a potentially-reentrant call, control also flows to a procedure outside \( C \).

The soundness of our treatment of invariants depends on the following soundness condition on analysis plugins.

**Condition 1** *(Set stationarity condition)* A set may only be modified by its defining module.

\(^5\)Initialization procedures can be modelled with a `init` boolean variable: `init = I` indicates that \( I \) holds after initialization. The developer would then use defaults to ensure that `init` is almost always true.

The flag, PALE and array analysis plugins presented in this paper satisfy the set stationarity condition. The following proposition is an immediate consequence of the set stationarity condition:

**Proposition 2** A set may only be modified when it is in scope.

We will prove that the invariant true at incoming boundary points by induction on instances of these points in program execution traces. This proof is in two parts: 1) for entry points of exported modules; and 2) for reentrant-call return sites. We first consider case 1), the incoming boundary points that occur as entry points of exported modules. Consider an arbitrary program execution trace \( s_0, s_1, \ldots, s_n \). At the first incoming boundary point \( s_{i_0} \), the invariant is true because it is true at \( s_0 \) and has not been changed since then (by the set stationarity condition). At subsequent incoming boundary points \( s_{i_k} \), the invariant will also be true, since an outgoing boundary point will have executed between \( s_{i_{k-1}} \) and \( s_{i_k} \) (recall we disallow calls to incoming boundary points from inside the scope, so that the program has to pass through an outgoing point), because the invariant was proven at \( s_{i_{k-1}} \), and because the truth of the invariant has not changed in the exterior code between the most recent outgoing boundary point and \( s_{i_k} \). Case 2), concerning reentrant-call incoming boundary points, is similar. Consider the sequence \( t_0, \ldots, t_n \) of points between the call \( t_0 \) and its return \( t_n \). The analysis explicitly proves the invariant at the outgoing boundary point \( t_0 \). If a trace has no incoming boundary point \( t_i \) between \( t_0 \) and \( t_n \), then the invariant still holds at \( t_n \), because of the set stationarity condition. For every incoming boundary point \( t_i \), then there must also exist an outgoing boundary point \( t_j \) (by Proposition 1) at which point the invariant is explicitly shown. Between the last outgoing boundary point and \( t_n \) (a subsequence which occurs outside the scope), the invariant is preserved, implying that it holds at \( t_n \).

**Consequences of Scopes** By using scopes, developers may omit details about transitive callees which are not relevant to understanding the effects of the caller. Furthermore, scope invariants allow the developer to assume that certain invariants always hold upon entry to the scope, which enhances the expressive power of our system.

### 4.2 Defaults for Simplifying Specifications by Omission

Many modules require that some initialization code be executed before normal operation of the module can proceed. Our system can represent this with an `Init`
boolean predicate attached to the appropriate module, and requiring that Init hold before (almost every) procedure in the module. Such a practice clutters procedure specifications with extra conjuncts.

We have created the notion of a default to address this problem. Defaults are named boolean clauses which are uniformly conjoined to requires and ensures clauses of procedure specifications unless they are explicitly suspended. Procedure p may declare a suspends clause; if default I is suspended in p, then the default is not applied to the requires and ensures clauses of p.

Defaults differ from scopes in that scopes talk about global cross-cutting concerns, whereas defaults talk about local properties specifically of interest to a particular specification, by freeing them from the obligation of specifying details of global interest in each procedure specification.

5 The Flag Plugin

Our flag analysis verifies that modules implement set specifications in which integer or boolean flags indicate abstract set membership. The developer specifies (using the flag abstraction language) the correspondence between concrete flag values and abstract sets from the specification, as well as the correspondence between the concrete and the abstract boolean variables. Figure 8 presents the syntax for our flag abstraction modules. This abstraction language defines abstract sets in two ways: (1) directly, by stating a base set; or (2) indirectly, as a set-algebraic combination of sets. Base sets have the form \( B = \{ x : T | x.f=c \} \) and include precisely the objects of type T whose field f has value c, where c is an integer or boolean constant; the analysis converts set algebra combinations of base sets, the flag analysis handles derived sets by conjoining the definitions of derived sets (in terms of base sets) to each verification condition and tracking the contents of the base sets. Derived sets may use named base sets in their definitions, but they may also use anonymous sets given by set comprehensions; the flag analysis assigns internal names to anonymous sets and tracks their values to compute the values of derived sets.

In our experience, applying several formula transformations drastically reduced the size of the formulas emitted by the flag analysis, as well as the time that the MONA decision procedure spent verifying these formulas. Section 5.4 describes these formula optimizations. These transformations greatly improved the performance of our analysis and allowed our analysis to verify larger programs.

5.1 Operation of the Analysis Algorithm

The flag analysis verifies a module \( M \) by sequentially checking each procedure of module \( M \). To verify a procedure, the analysis performs abstract interpretation [10] with analysis domain elements represented by formulas. Our analysis associates quantified boolean formulas \( B \) to each program point. A formula \( F \) has two collections of set variables: unprimed set variables \( S \) denoting initial values of sets at the entry point of the procedure, and primed set variables \( S' \) denoting the values of these sets at the current program point; \( F \) also contains unprimed and primed boolean variables \( b \) and \( b' \) representing the pre- and post-values of local and global boolean variables. The interpretations of these variables are given by the definitions in the abstraction section of the module. The use of primed and unprimed variables allows our analysis to represent, for each program point \( p \), a binary relation on states that overapproximates the reachability relation between procedure entry and program point \( p \) [11, 17, 32].

In addition to the abstract sets from the specification, the analysis also generates a set for each (object-typed) local variable. This set contains the object to which the local variable refers and has a cardinality constraint that restricts the set to have cardinality at most one (the empty set represents a null reference).

The formulas that the analysis manipulates therefore support the disambiguation of local variable and object field accesses at the granularity of the sets in the analysis; other analyses often rely on a separate pointer analysis to provide this information.

The initial dataflow fact at the start of a procedure is the precondition for that procedure, transformed into a relation by conjoining \( S' = S \) for all relevant sets. At merge points, the analysis uses disjunction to combine boolean formulas. Our current analysis iterates while loops at most some constant number of times, then coarsens the formula to true to ensure termination, thus applying a simple form of widening [10]. The analysis also allows the developer to provide loop in-
variants directly. After running the dataflow analysis, our analysis checks that the procedure conforms to its specification by checking that the derived postcondition (which includes the \textit{ensures} clause and any required representation or global invariants) holds at all exit points of the procedure. In particular, the flag analysis checks that for each exit point \( e \), the computed formula \( B_e \) implies the procedure’s postcondition.

\textbf{Incorporation.} The transfer functions in the dataflow analysis update boolean formulas to reflect the effect of each statement. Recall that the dataflow facts for the analysis are boolean formulas representing the relation between the state at procedure entry and the state at the current program point. Let \( B_s \) be the boolean formula describing the effect of statement \( s \). The incorporation operation \( B \circ B_s \) is the result of symbolically computing the relation composition of relations given by formulas \( B \) and \( B_s \). Conceptually, incorporation updates \( B \) with the effect of \( B_s \). We compute \( B \circ B_s \) by applying equivalence-preserving simplifications to the formula

\[
\exists \hat{S}_1, \dot{\ldots} \hat{S}_n. \ B[S'_1 \mapsto \hat{S}_1] \land B_s[S \mapsto \hat{S}_1]
\]

5.2 \textbf{Transfer Functions}

Our flag analysis handles each statement in the implementation language by providing appropriate transfer functions for these statements. The generic transfer function is a relation of the following form:

\[
\lbrack\text{st}\rbrack(B) \equiv B \circ \mathcal{F}(\text{st})
\]

where \( \mathcal{F}(\text{st}) \) is the formula symbolically representing the transition relation for statement \( \text{st} \) expressed in terms of abstract sets. The transition relations for the statements in our implementation language are as follows.

\textbf{Assignment statements.} We first define a generic frame condition generator, used in our transfer functions,

\[
\text{fram}_x = \bigwedge_{s \neq x, \text{not derived}} \bigwedge_{p' \neq p} (s' = s \land (p' \Leftrightarrow p)),
\]

where \( s \) ranges over sets and \( p \) over boolean predicates. Note that derived sets are not preserved by frame conditions; instead, the analysis preserves the anonymous sets contained in the derived set definitions and joins these definitions to formulas before applying the decision procedure.

Our flag analysis also tracks values of boolean variables:

\[
\begin{align*}
\mathcal{F}(\text{b = true}) &= \ b' \land \text{fram}_b \\
\mathcal{F}(\text{b = false}) &= \ \lnot b' \land \text{fram}_b \\
\mathcal{F}(\text{b = y}) &= \ (b' \Leftrightarrow y) \land \text{fram}_b \\
\mathcal{F}(\text{if (cond))} &= \ (b' \Rightarrow \mathcal{F}'(\text{if (cond)})) \land \text{fram}_b \\
\mathcal{F}(\text{b = e}) &= \ (b' \Leftrightarrow \lnot e) \land \text{fram}_b
\end{align*}
\]

where \( f^+(e) \) is the result of evaluating the condition \( e \), as defined below in our analysis of if statements.

We also track the local variable object references:

\[
\begin{align*}
\mathcal{F}(\text{x = y}) &= \ (x' = y') \land \text{fram}_x \\
\mathcal{F}(\text{x = null}) &= \ (x' = \emptyset) \land \text{fram}_x \\
\mathcal{F}(\text{x = new i}) &= \ (x' = \emptyset) \land \bigwedge_{x \in S}(x' \cap S = \emptyset) \land \text{fram}_x
\end{align*}
\]

We next present the transfer function for changing set membership. If \( R = \{x : T \mid x.f = e\} \) is a set definition in the abstraction section, we have:

\[
\mathcal{F}(x.f = c) := R' = R \cup x \land \bigwedge_{S \in \text{alts}(R)} S' = S \setminus x \land \text{fram}_{(R) \cup \text{alts}(R)}
\]

where \( \text{alts}(R) := R' \) such that the abstraction module contains \( R' = \{x : T \mid x.f = c_1\}, c_1 \neq c \).

We also have a rule handling field reads and writes of boolean values \( b \); it is similar to the rule above for reads and writes of integers. However, since our analysis tracks the flow of boolean values, the rules are more detailed. When \( B^+ = \{x : T \mid x.f = \text{true}\} \) and \( B^- = \{x : T \mid x.f = \text{false}\} \), the rule is:

\[
\begin{align*}
\mathcal{F}(x.f = b) &= \ \bigl( b \land B^+ = B^+ \cup x \\
& \quad \land \bigwedge_{S \in \text{alts}(B^+)} S' = S \setminus x \\
& \quad \land \bigl( \lnot b \land B^- = B^- \cup x \\
& \quad \land \bigwedge_{S \in \text{alts}(B^-)} S' = S \setminus x \bigr) \bigr) \\
\mathcal{F}(b = y.f) &= \ (b' \Leftrightarrow \lnot \in B^+ \land \text{fram}_b
\end{align*}
\]

Finally, we have some default rules to conservatively account for expressions not otherwise handled,

\[
\mathcal{F}(x.f = *) = \text{fram}_x \quad \mathcal{F}(x = *) = \text{fram}_x.
\]

\textbf{Procedure calls.} For a procedure call \( \text{x=proc(y)} \), our transfer function checks that the callee’s requires condition holds, and incorporates \text{proc}’s ensures condition as follows:

\[
\mathcal{F}(x = \text{proc}(y)) = \text{ensures}_1(\text{proc}) \land \bigwedge_S S = S
\]

where both \text{ensures}_1 and \text{requires}_1 substitute caller actuals for formals of \text{proc} (including the return value), and where \( S \) ranges over all local variables.

\textbf{Conditionals.} The analysis produces a different formula for each branch of an if statement \( \text{if (e)} \). We define functions \( f^+(e), f^-(e) \) to summarize the additional information available on each branch of the conditional; the transfer functions for the true and false branches of the conditional are thus, respectively,

\[
\begin{align*}
\lbrack\text{if (e)}\rbrack^+(B) &= f^+(e) \land B \\
\lbrack\text{if (e)}\rbrack^-(B) &= f^-(e) \land B
\end{align*}
\]
For constants and logical operations, we define the obvious \( f^+ \), \( f^- \):

\[
\begin{align*}
  f^+(\text{true}) &= \text{true} & f^-(\text{true}) &= \text{false} \\
  f^+(\text{false}) &= \text{false} & f^-(\text{false}) &= \text{true} \\
  f^+(\text{lc}) &= f^-(\text{lc}) & f^+(\text{lc}) &= f^-(\text{lc}) \\
  f^+(\text{eq}) &= f^-(\text{eq}) & f^+(\text{eq}) &= f^-(\text{eq}) \\
  f^+(\text{eq}) &= f^+(x=\text{eq}) & f^-(\text{eq}) &= f^+(x=\text{eq})
\end{align*}
\]

We define \( f^+, f^- \) for boolean fields as follows:

\[
\begin{align*}
  f^+(e_1 \&\& e_2) &= f^+(e_1) \&\& f^+(e_2) \\
  f^-(e_1 \&\& e_2) &= f^-(e_1) \&\& f^-(e_2)
\end{align*}
\]

for boolean fields as follows:

\[
\begin{align*}
  f^+(e) &= x \subseteq B \\
  f^-(e) &= x \nsubseteq B \\
  f^+(e=false) &= x \nsubseteq B \\
  f^-(e=false) &= x \subseteq B
\end{align*}
\]

where \( B = \{ x : T \mid x.f = \text{true} \} \); analogously, let \( R = \{ x : T \mid x.f = \text{c} \} \). Then,

\[
\begin{align*}
  f^+(x.f=\text{c}) &= x \subseteq R \\
  f^-(x.f=\text{c}) &= x \nsubseteq R.
\end{align*}
\]

We also predicate the analysis on whether a reference is null or not:

\[
\begin{align*}
  f^+(x=\text{null}) &= x = \emptyset \\
  f^-(x=\text{null}) &= x \neq \emptyset.
\end{align*}
\]

Finally, we have a catch-all condition,

\[
\begin{align*}
  f^+(\_\_) &= \text{true} & f^-(\_\_) &= \text{true}
\end{align*}
\]

which conservatively captures the effect of unknown conditions.

**Loops.** Our analysis analyzes while statements by synthesizing loop invariants or by verifying developer-provided loop invariants. To synthesize a loop invariant, it iterates the analysis of the loop body until it reaches a fixed point, or until \( N \) iterations have occurred (in which case it synthesizes true). The conditional at the top of the loop is analyzed the same way if statements are analyzed. We can also verify explicit loop invariants; these simplify the analysis of while loops and allow the analysis to avoid the fixed point computation involved in deriving a loop invariant. Developer-supplied explicit loop invariants are automatically conjoined with the frame conditions generated by the containing procedure’s modifies clause to ease the burden on the developer.

**Assertions and Assume Statements.** We analyze statement \( s \) of the form assert \( \mathcal{A} \) by showing that the formula for the program point \( s \) implies \( \mathcal{A} \). Assertions allow developers to check that a given set-based property holds at an intermediate point of a procedure. Using assume statements, we allow the developer to specify properties that are known to be true, but which have not been shown to hold by this analysis. Our analysis prints out a warning message when it processes assume statements, and conjoins the assumption to the current dataflow fact. Assume statements have proven to be valuable in understanding the analysis outcomes during the debugging of procedure specifications and implementations. Assume statements may also be used to communicate properties of the implementation that go beyond the abstract representation used by the analysis.

**Return Statements.** Our analysis processes the statement return \( x \) as an assignment \( rv = x \), where \( rv \) is the name given to the return value in the procedure declaration. For all return statements (whether or not a value is returned), our analysis checks that the current formula implies the procedure’s postcondition and stops propagating that formula through the procedure.

### 5.3 Verifying Implication of Dataflow Facts

A compositional program analysis needs to verify implication of constraints as part of its operation. Our flag analysis verifies implication when it encounters an assertion, procedure call, or procedure postcondition. In these situations, the analysis generates a formula of the form \( B \Rightarrow A \) where \( B \) is the current dataflow fact and \( A \) is the claim to be verified\(^7\). The implication to be verified, \( B \Rightarrow A \), is a formula in the boolean algebra of sets, and we check its validity using the MONA decision procedure for the monadic second-order logic of strings, which subsumes boolean algebras [18].

### 5.4 Boolean Algebra Formula Transformations

In our experience, applying several formula transformations drastically reduced the size of the formulas emitted by the flag analysis, as well as the time needed to determine their validity using an external decision procedure; in fact, some benchmarks could only be verified with the formula transformations enabled. This subsection describes the transformations we found to be useful and includes a performance evaluation of these transformations, comparing formula sizes and analysis running times.

**Smart Constructors.** The constructors for creating boolean algebra formulas apply peephole transformations as formulas are being created. The simplest peephole transformation is constant folding: for instance, attempting to create \( B \&\& \text{true} \) gives the formula \( B \). Our constructors fold constants in implications, conjunctions, disjunctions, and negations. Similarly, attempting to quantify over unused variables causes the quantifier to be dropped: \( \exists x.F \) is created as just \( F \) for \( x \) not free in \( F \). Most interest-

---

\(^7\) Note that \( B \) may be unsatisfiable; this often indicates a problem with the program’s specification. The flag analysis can, optionally, check whether \( B \) is unsatisfiable and emit a warning if it is. This check enabled us to improve the quality of our specifications by identifying specifications that were simply incorrect.
ingly, we factor common conjuncts out of disjunctions: 
\((A \land B) \lor (A \land C)\) is represented as 
\(A \land (B \lor C)\). Conjunct
factoring greatly reduces the size of formulas tracked af-
fter control-flow merges, since most conjuncts are shared
on both control-flow branches. The effects of this trans-
formations appear similar to the effects of the SSA form
conversion in weakest precondition computation [14,27].

Basic Quantifier Elimination. We symbolically
compute the composition of statement relations dur-
ing the incorporation step by existentially quantifying
over all state variables. However, most relations cor-
responding to statements modify only a small part of
the state and contain the frame condition that indicates
that the rest of the state is preserved. The result of in-
corporation can therefore often be written in the form
\(\exists x_1. x = x_1 \land F(x),\) which is equivalent to
\(F(x_1)\). In this way we reduce both the number of conjuncts and the
number of quantifiers. Moreover, this transformation
can reduce some conjuncts to the form 
\(t = t\) for some
Boolean algebra term \(t\), which is a true conjunct that
is eliminated by further simplifications.

It is instructive to compare our technique to weakest
precondition computation [14] and forward symbolic ex-
ecution [9]. These techniques are optimized for the com-
mon case of assignment statements and perform relation
composition and quantifier elimination in one step. Our
technique achieves the same result, but is methodolog-
ically simpler and applies more generally. In particu-
ar, our technique can take advantage of equalities in
transfer functions that are not a result of analyzing as-
signment statements, but are given by explicit formulas
in \texttt{ensures} clauses of procedure specifications. Such
transfer functions may specify more general equalities
such as \(A = A' \cup x \land B' = B \cup x\) which do not reduce to
simple backward or forward substitution.

Quantifier Nesting. We have experimentally ob-
served that the MONA decision procedure works sub-
stantially faster when each quantifier is applied to the
smallest scope possible. We have therefore implemented
a quantifier nesting step that reduces the scope of each
quantifier to the smallest possible subformula that con-
tains all free variables in the scope of the quantifier.
For example, our transformation replaces the formula
\(\forall x. \forall y. (f(x) \Rightarrow g(y))\) with \((\exists x. f(x)) \Rightarrow (\forall y. g(y))\).

To take maximal advantage of our transformations,
we simplify the formula after applying incorporation
and before invoking the decision procedure. Our global
simplification step rebuilds the formula bottom-up and
applies the simplifications to each subformula.

5.5 Evaluating the Impact of Formula Trans-
formation

Table 1 shows the result of our formula transformations. The
\texttt{compiler} benchmark models a constant-folding
compiler pass. The \texttt{scheduler} benchmark models an
operating system scheduler. The \texttt{ctas} benchmark is
the core of an air-traffic control system. The \texttt{board},
controller and \texttt{view} modules are the core modules of the
\texttt{minesweeper} example.

We ran our benchmarks on a Pentium 4 at 2.80GHz,
running Linux, with 2 gigabytes of RAM. We have
reported the sizes (in terms of AST node counts) of
the boolean algebra formulas created with all transforma-
tions enabled; with all transformations except for
smart constructors; and with no transformations en-
dabled. (The results with smart constructors and no
other transformations were usually identical to the re-
sults with no transformations.) For each run, we have
also presented the time spent in the decision procedure
(under 4 seconds, optimized) and in the analysis, ex-
cluding the decision procedure (under 25 seconds, op-
timized). Our formula transformations reduce formula
size by 2 to 70 times (with greater reductions for larger
formulas); indeed, without transformation, the formu-
las generated by \texttt{compiler}, \texttt{board} and \texttt{view} could not
successfully be checked by MONA because of an out of
memory error.

<table>
<thead>
<tr>
<th></th>
<th>Opt</th>
<th>Smart Constrs</th>
<th># nodes</th>
<th>MONA time</th>
<th>Flag time</th>
<th>Opt. ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler</td>
<td>✓</td>
<td>✓</td>
<td>15800</td>
<td>0.45</td>
<td>7.84</td>
<td>38.36</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>×</td>
<td>28000</td>
<td>0.60</td>
<td>9.68</td>
<td>21.72</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>✓</td>
<td>608375</td>
<td>N/A</td>
<td>82.04</td>
<td>1.00</td>
</tr>
<tr>
<td>scheduler</td>
<td>✓</td>
<td>✓</td>
<td>468</td>
<td>0.05</td>
<td>0.04</td>
<td>3.92</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>✓</td>
<td>1086</td>
<td>0.08</td>
<td>0.04</td>
<td>3.92</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>×</td>
<td>3410</td>
<td>0.23</td>
<td>0.11</td>
<td>2.15</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>✓</td>
<td>9726</td>
<td>13.33</td>
<td>0.29</td>
<td>1.00</td>
</tr>
<tr>
<td>ctas</td>
<td>✓</td>
<td>✓</td>
<td>468</td>
<td>0.05</td>
<td>0.04</td>
<td>3.92</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>×</td>
<td>1086</td>
<td>0.08</td>
<td>0.04</td>
<td>3.92</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>×</td>
<td>3410</td>
<td>0.23</td>
<td>0.11</td>
<td>2.15</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>✓</td>
<td>9726</td>
<td>13.33</td>
<td>0.29</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>101872</td>
<td>4.45</td>
<td>44.56</td>
<td>17.60</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>×</td>
<td>1793295</td>
<td>N/A</td>
<td>369.90</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 1: Formula sizes before and after transformation

6 The PALE Analysis Plugin

Unlike the flag analysis, which we designed to operate
within our analysis framework, the PALE analysis is
a previously implemented analysis package that we in-
tegrated into our framework. During the course of this
adaptation, we did not modify the PALE analysis pack-
age itself — we instead implemented translators that enabled it to work within our analysis framework.

6.1 The PALE Analysis System

The PALE analysis system takes as input a program written in its own imperative language [30]. This program includes preconditions, postconditions, loop invariants, and graph type declarations [20]. A graph type is a tree-like pointer-based (potentially recursive) data structure with a distinguished set of data fields (such as the next field in Figure 9),\(^8\) whose values form the spanning tree backbone of the data structure. In addition to data fields, a graph type may contain routing fields [20] (such as the prev field in Figure 9). These routing fields are functionally determined by the backbone; the prev field in Figure 9, for example, is uniquely determined as the inverse of the next field. By identifying data fields that form the spanning tree and by providing the definitions for the derived fields, graph type declarations allow the developer to specify the representation invariants that the data structures must satisfy.

\[
\text{abst module DLLSet} \{ \\
\quad \text{use plugin "PALE";} \\
\quad \text{Content } = \{x : \text{Entry} \mid \text{"root<next*>x"};\} \\
\quad \text{invariant "type Entry } = \{ \\
\quad \quad \text{data next : Entry; } \\
\quad \quad \text{pointer prev:Entry[this"Entry.next = \{prev\}]; } \\
\quad \quad \text{invariant "data root:Entry;";} \\
\quad \}
\]

Figure 9: Doubly-Linked List Abstraction Section

The precondition, postcondition, and loop invariants are arbitrary formulas in monadic second-order logic. Such formulas enable the use of transitive closure over object reference fields to identify the set of all objects that participate in that data structure. Building on this base, it is also possible to specify arbitrary boolean formulas containing set inclusion and equality constraints involving these sets. For example, it is possible to specify that the sets of objects in two lists (identified using transitive closure over the next field) are disjoint, equal, or that one is a subset of the other. It is also possible to state set membership constraints involving the objects that variables point to.

The PALE analysis system translates an input program into a collection of verification conditions whose validity guarantees that the procedures in the program satisfy their precondition/loop invariant/postcondition relationships. These verification conditions are formulas in monadic second-order logic. The PALE system uses the MONA decision procedure [18, 19] to determine the validity of these verification conditions. If all of these conditions are valid, the program satisfies its PALE specification.

6.2 Using the PALE Plugin

We next describe the information that the developer provides to enable the PALE plugin to verify that a module implementation conforms to its specification. Most of the information specific to the PALE plugin is contained in the abstraction section, whose syntax is in Figure 10.

6.2.1 Specifying Set Definitions

The developer specifies the abstraction function for a graph type data structure by defining the content of an abstract set using a formula in monadic second-order logic. Figure 9 shows a definition of the set Content as the set of all Entry objects reachable from the root along the next field. A binary relation given by a regular expression such as <next*> is a shorthand for the corresponding formula with two variables definable in monadic second-order logic.

6.2.2 Specifying Representation Invariants

The developer specifies the representation invariants for the PALE plugin using invariant declarations in the abstraction section, as illustrated in Figure 9. The syntax of these invariants is specific to the PALE plugin. An invariant for the PALE plugin is either a graph type definition, such as the definition of the Entry graph type in Figure 9, or a declaration of a data structure root, such as the data root:Entry declaration in Figure 9.

These representation invariants impose the following constraint on the heap: each object is either 1) a member of the data structure or 2) an object external to the data structure. Each member object is reachable from the data structure root along the data fields. In addition to data fields, a member object has routing fields (denoted by the pointer keyword) whose value is given by the formula specified in the graph type definition. On the other hand, each external object is unreachable.

---

\(^8\)Note that, in the PALE system terminology, “data” fields hold reference values.
6.3 Translation to PALE Input Language

We incorporated the PALE analysis system into our pluggable analysis framework by 1) using abstraction sections to translate our common set-based specifications into PALE specifications, 2) translating statements into the imperative language accepted by PALE, and 3) translating loop invariants into PALE loop invariants. The loop invariants in implementation modules verified by the PALE plugin contain two parts. The first part contains concrete data structure properties, and is literally transferred into the PALE implementation language. The second part contains abstract set properties, and is translated in the same way that the requires and ensures clauses are translated. Our translation also elides integer variables from the input program; integer variables are not supported by the PALE input language.

For each set definition of the form

\[ S = \{ x : T \mid F(x) \} \]

that appears in the abstraction section, the translator produces a second-order predicate that takes a set as an argument:

\texttt{pale isS(set S:T) = allpos x of T: x in S => F(x)}

A statement \( B(S_1, \ldots, S_n) \) in boolean algebra of sets then corresponds to the formula

\[ \exists S_1, \ldots, S_n. \bigwedge_{i=1}^n \text{isS}_i(S_i) \land B(S_1, \ldots, S_n) \]

The translator \texttt{isS}_i predicates to translate the specification of a procedure \( p \) as follows. Consider a specification of the form

\begin{align*}
\text{requires } & B_0(S_1, \ldots, S_n) \\
\text{modifies } & S_{j_1}, \ldots, S_{j_m} \\
\text{ensures } & B_1(S_1, \ldots, S_n, S'_1, \ldots, S'_l) \\
\end{align*}

The first translation step eliminates the modifies clause, yielding

\begin{align*}
\text{requires } & B_0(S_1, \ldots, S_n) \\
\text{ensures } & B_2(S_1, \ldots, S_n, S'_1, \ldots, S'_l) \\
\end{align*}

where

\[ B_2 = B_1 \land \bigwedge_{i \notin \{j_1, \ldots, j_m\}} S'_i = S_i \]

The next translation step introduces logical variables \( S_1, \ldots, S_n \) that correlate preconditions and postconditions. The resulting precondition/postcondition pair is:

\begin{verbatim}
set S_1 : T_1; 
... 
set S_n : T_n; /* precondition */ 
[ \bigwedge_{i=1}^n \text{isS}_i(S_i) \land B_0(S_1, \ldots, S_n)] 
{stmts} 
/* postcondition */ 
[ \exists \text{set } S'_1 \text{ of } T_1 : \text{isS}_i(S'_1) \land 
... 
\exists \text{set } S'_n \text{ of } T_n : \text{isS}_n(S'_n) \land 
B_2(S_1, \ldots, S_n, S'_1, \ldots, S'_l) ]
\end{verbatim}

where \{\texttt{stmts}\} is the translation of the statements that implement the body of the procedure \( p \). At this point we have a translated procedure that we can pass to the PALE analysis system for verification. Given a module to verify, our analysis driver translates all of the procedures into this form and checks if the PALE system can verify them. If so, the implementation section correctly implements its specification.

6.4 Implications

The PALE analysis package implements a sophisticated analysis that can verify detailed properties of complex linked data structures. It is clearly infeasible (for scalability reasons) to use PALE to analyze anything other than encapsulated data structure implementations. But within this domain it can provide exceptional precision and verify important properties that are clearly beyond the reach of more scalable analyses.

Our successful integration of the PALE analysis system demonstrates that it is possible to apply very precise analyses to focused parts of the program. Our results therefore show how to unlock the potential of these analyses to verify important data structure consistency properties in programs that would otherwise remain beyond reach.

7 The Array Analysis Plugin

The array analysis plugin generates verification conditions using weakest preconditions and discharges them using the Isabelle theorem prover. We have chosen this technique as a last resort for verifying arbitrarily complicated data structure implementations. The logic for specifying abstraction functions is based on typed set
theory and proof obligations can be discharged using automated theorem proving or a proof checker for manually generated proofs, which means that there is no a priori bound on the complexity of the data structures (and data structure consistency properties) that can be verified. In our current implementation we have explored this technique for data structures that implement sets by storing objects in global arrays. For example, we have verified the operations on abstract set

\[ \text{Content} = \{ x | \exists j. 0 \leq j \wedge j < s \wedge x \in d[j] \} \]

where \( d \) is a global array of objects and \( s \) is an integer variable indicating the currently used part of the array.

The plugin analyzes each procedure independently, showing that it conforms to its specification using the following phases:

1. Concretization: Implicitly conjoin each postcondition with the frame condition derived from modifies clauses. Apply the definitions of sets from the abstraction section to preconditions and postconditions in specification sections, as well as loop invariants and assertions. The result are conditions expressed in terms of the concrete data structure state. For example, the postcondition \( \text{Content}' = \text{Content} - e \) translates into the formula

\[ \{ x | \exists j. 0 \leq j \wedge j < s' \wedge x \in d'[j] \} = \{ x | \exists j. 0 \leq j \wedge j < s \wedge x \in d[j] \} - e \]

2. Representation invariants: Conjoin both precondition and postcondition with representation invariants specified in the abstraction section. In our example we need a representation invariant \( 0 \leq s \).

3. Statement desugaring: translate statements into loop-free guarded command language (e.g. [14]).

4. Verification condition generation: using weakest precondition semantics, create the formula whose validity implies the conformance of the procedure with respect to its specification.

5. Separation: Separate the verification condition into as many conjuncts as possible by performing a simple non-backtracking natural-deduction search through connectives \( \forall, \Rightarrow, \land \).

6. Verification: Attempt to verify each conjunct in turn. Verify if the conjunct is in the library of proved lemmas; if not, attempt to discharge it using the proof hint supplied in procedure code; if no hint is supplied, invoke the Isabelle's built-in simplifier and classical reasoner with array axioms.

In our example, most of the generated verification-condition conjuncts are discharged automatically using array axioms. For the remaining ones, the fully automated verification fails and they are printed as “not known to be true”. After interactively proving these difficult cases in Isabelle, they are stored in the library of verified lemmas and the subsequent verification attempts pass successfully without assistance.

8 Experience

We implemented our system and, to obtain experience using it, coded up several benchmark programs, using our system during the development of the programs. In addition to the minesweeper example presented in Section 2, we ran our analysis on programs with computational patterns from scientific computations, operating-system schedulers, air-traffic control, and program transformation passes. These benchmarks use a variety of data structures, and we have therefore implemented and verified sets, set iterators, queues, stacks, and priority queues. Table 2 illustrates the benchmarks we ran through our system. Our implementations range from singly-linked and doubly-linked lists and tree insertion (all verified using the PALE plugin) through array data structures (verified using the array membership plugin with the Isabelle theorem prover used to discharge verification conditions).

Implementation Structure. Our implementation provides an infrastructure with several general components that perform tasks required by all analyses. The implementation language component can parse and type-check implementation sections. It produces an abstract syntax tree and methods that allow analyses to conveniently access this representation. Similarly, the specification component can parse and type check specification sections and provides access to the resulting abstract syntax tree. Large parts of abstraction sections are expressed in a language that is specific to each analysis. The abstraction section component parses those parts of the abstraction section syntax that are common to all analyses and uses uninterpreted strings to pass along the analysis-specific parts. Finally, the implementation provides a driver that processes the program and invokes the appropriate analysis for each module that it encounters. Our implementation consists of approximately 10,000 lines of O'Caml code, to which the flag plugin contributes 2000 lines, the PALE plugin another 700 lines, and the array analysis plugin 1000 lines.

Because we implemented the flag analysis specifically for this project, it is fairly closely integrated with the rest of our infrastructure. It processes implementation and specification section directly in the abstract syntax tree representation that our infrastructure provides. The PALE analysis plugin, on the other hand, uses an off-the-shelf analysis package that was developed before the start of this project. We therefore wrote an adapter that integrates this analysis into our system, as described in Section 6.3. The array analysis plugin reads the specification, abstraction and implementation sections and produces proof obligations, using weakest preconditions, and discharges them using the Isabelle theorem prover. The developer may specify proof hints, on a per-procedure basis, that invoke arbitrarily complicated previously-proved lemmas.

### 8.1 Minesweeper

We earlier described some of the invariants that we successfully verify for the minesweeper example. While we were trying to verify our invariants about the minesweeper implementation, we found a number of bugs in that implementation. We now present one of the bugs that we found. The situation is that at the end of the game, minesweeper exposes the entire game board; we use `removeFirst` to remove all elements from the unexposed list, one at a time. After we have exposed the entire board, we can guarantee that the list of unexposed cells is empty:

```plaintext
proc drawFieldEnd()
    requires ExposedList.setInit & Board.gameOver &
        (UnexposedList.Content <= Board.U)
    modifies UnexposedList.Content, Board.ExposedCells,
        Board.UnexposedCells, ExposedList.Content,
        UnexposedList.Content
    ensures card(UnexposedList.Content') = 0;
```

because the implementation of the `drawFieldEnd` procedure loops until `isEmpty` returns `true`, which also guarantees that the `UnexposedList.Content` set is empty.

The natural way to write the iteration in this procedure would be:

```plaintext
while (UnexposedList.isEmpty()) {
    Cell c = UnexposedList.removeFirst();
    drawCellEnd(c);
}
```

and indeed, this was the initial implementation of that code. However, when we attempted to analyze this code, we got the following error message:

Analyzing proc drawFieldEnd...
Error found analyzing procedure drawFieldEnd:
    requires clause in a call to procedure View.drawCellEnd.
```

Upon further examination, we found that we were breaking the invariant ensuring that `Board.ExposedCells.equals(UnexposedList.Content)`. The correct way to preserve the invariant is by calling `Board.setExposed`, which simultaneously sets the `isExposed` flag and removes the cell from the `UnexposedList`:

```plaintext
Cell c = UnexposedList.getFirst();
Board.setExposed(c, true);
drawCellEnd(c);
```

which successfully analyzes:

Analyzing proc drawFieldEnd... Procedure drawFieldEnd passes.

### 8.2 Stack Data Structure

Using our system, we have implemented stacks, queues, priority queues, sets, and iterators using singly-linked lists, doubly-linked lists and trees. We checked these implementations with the PALE plugin. It turns out that our initial implementations were not completely correct; our analysis pinpointed (and helped us correct) some errors in the implementations. We report, below, two bugs that were found by our PALE plugin.

For the stack, we maintain an abstract set `S` representing the content of the stack, and verify that
stack insertions actually insert the given object into the stack $(S' = S + e)$, and that removal actually removes an object from the stack, if possible: \( \text{card}(S) = 0 \land (\exists e : \text{Entry}. \ (S' = S - e) \land \text{card}(e) = 1) \).

Our PALE plugin checks that objects that belong to a set have consistent values for navigational fields (e.g. \texttt{next, prev}), and that objects that do not belong to the set have those fields set to \texttt{null}. Initially, our implementation for \texttt{removeFirst} was:

```cpp
proc removeFirst() returns e : Entry {
  Entry res = root;
  if (root != null) root = root.next;
  pragma "removed res";
  return res;
}
```

where the \texttt{pragma} statement indicates to the analysis that it is verifying a set removal. However, the analysis reports an error while verifying this implementation. Careful inspection of this code, however, reveals that the removed object, \texttt{res}, will still have a reference to an object in the stack after removal; this is potentially problematic, as it may lead to non-list structures being present in the heap. Our plugin therefore requires us to add \texttt{res.next = null} to this procedure, so that all objects subsequently passed to this module will have \texttt{next} set to \texttt{null}.

### 8.3 List Iterators

We have implemented (using a singly-linked list) a set which supports iterators; it has a procedure which returns the next element in the set, until there are no more elements. We have modelled this set using a module which declares two sets, \texttt{Content} and \texttt{Iter}; the \texttt{Iter} set contains all elements which have not yet been returned by the iterator. Note that we can guarantee, by reasoning solely at the abstract set level, that the \texttt{nextIter} procedure returns every member of the \texttt{Content} set.

Our iterable set implementation, however, also supports removal. In the presence of iteration, removal has the following semantics: if we remove an object in \texttt{Iter}, then it will not be returned by subsequent calls to the iterator; if the object is not in \texttt{Iter}, then future iterations are unchanged. Our analysis caught the corner case where we remove the element which would next be returned by the iterator. Adding the following line allowed the analysis of this module to succeed:

```cpp
if (current == e) current = current.next;
```

### 8.4 Program Transformations

This benchmark implements a constant-folding operation on the intermediate representation of a simple programming language. The input to this operation is an abstract syntax tree along with a list of all nodes in the tree. The output is a new tree after constant folding (this transformation replaces arithmetic expressions on constant values with the computed value of the arithmetic expression). To facilitate memory management, each tree comes with a list of nodes in the tree.

For efficiency, the transformation should, whenever possible, reuse (instead of copy) nodes from the input tree when it constructs the output tree. An implementation may therefore fail to remove reused nodes from the input tree’s list, leading to premature deallocation and data structure corruption. In our system, the program eliminates this possibility by using global invariants to require the sets of nodes in the input tree list and output tree list to be disjoint. An additional global invariant requires the output tree list to contain all of the nodes in the transformed output tree.

### 8.5 Process Scheduler

Our process scheduler benchmark maintains a list of running processes and a priority queue of suspended processes. There are three modules in our process scheduler implementation: the \texttt{RunningList} module (which maintains the list of running processes), the \texttt{SuspendedQueue} module (which maintains the queue of suspended processes), and the \texttt{Scheduler} module (which implements the specification for the scheduler). The running list and suspended queue are verified using the PALE plugin, whereas the scheduler itself is verified with the flag plugin. Both the data structures and the core scheduler know whether a process is running or suspended: the core scheduler uses flags to indicate set membership, whereas the data structures use heap reachability to track membership. One of the global invariants ensures that the sets in the scheduler and in the data structures coincide. Our analysis also verifies that the set of running processes is always disjoint from the set of suspended processes:

```cpp
invariant (Running = RunningList.InList) \land
(Suspended = SuspendedQueue.InQueue) \land
disjoint (Running, Suspended);
```

### 8.6 CTAS

The Center-TRACON Automation System (CTAS) is a set of air-traffic control tools developed at the NASA Ames research center [1]. The system is designed to help air traffic controllers visualize and manage the complex air traffic flows at centers surrounding large metropolitan airports. CTAS is structured with a central communications manager process that maintains socket connections to a graphics process, a weather process (to acquire information about the weather), a track acquisition process (to acquire radar data), and a trajectory control process.
synthesizer (to compute predicted trajectories for the controlled aircraft).

The weather and track acquisition sockets are read-only: the communications manager simply acquires the data that they send. The graphics socket, on the other hand, is write-only: the job of the graphics process is to display the information to the control. The communications manager both reads and writes the trajectory socket: it writes the socket to send requests to project the trajectory for the controlled aircraft and reads the socket to obtain the synthesized trajectories.

We implemented a program with this communication pattern and used our system to check these access constraints. The final system ensures that all sockets are correctly initialized (and have not been closed) when the program attempts to read or write to or from the socket. Our sets include a set of writable sockets, a set of readable sockets, and a set of closed sockets. The weather and track acquisition sockets are elements of the set of readable sockets only, the graphics socket is an element of the set of writable sockets only, while the trajectory socket is an element of both sets. Note that enabling a socket to participate in multiple sets at the same time (in effect, composing the typestate out of multiple orthogonal sets) substantially simplifies the resulting typestate system. A typical example of an requires clause is that for the readTrack procedure:

```plaintext
proc readTrack() requires card(Track.Data)=1 &
(Track.Data in Sockets.Open) &
(Track.Data in Sockets.Readable)
ensures true;
```

The preconditions of the socket read and write procedures require the socket to be in the read or write set, respectively, ensuring that the program does not attempt to perform an inappropriate socket operation.

### 8.7 Water

Our Water benchmark is a port of the Perfect Club benchmark MDG [5] to our implementation language. It uses a predictor/corrector method to evaluate forces and potentials in a system of water molecules in the liquid state. The central loop of the computation performs a time step simulation. Each step predicts the state of the simulation, uses the predicted state to compute the forces acting on each molecule, uses the computed forces to correct the prediction and obtain a new simulation state, and uses the new simulation state to compute the potential and kinetic energy of the system.

The Water benchmark consists of several modules, including the simparm, atom, H2O, ensemble, and main modules. These modules contain 2000 lines of implementation and 500 lines of specification. Each module defines sets and boolean variables; we use these sets and variables to express safety properties about the computation.

The simparm module, for instance, is responsible for recording the simulation parameters, which are stored in a text file and loaded upon demand. This module therefore defines two boolean variables, Init and ParmsLoaded; Init implies that the module has been initialized, i.e. the appropriate arrays have been allocated on the heap, while the ParmsLoaded variable implies that the simulation parameters have been loaded from disk. Our analysis verifies that no simulation parameter may be requested until the parameters have been loaded.

The fundamental unit of the simulation is the atom, which is encapsulated by the atom module; atoms may be predicted or corrected, and the predc and corre procedures change atoms into predicted or corrected atoms if the appropriate preconditions are met. In particular, the simulation may only correct a predicted atom; to enforce this property in the specification, we define sets Predic and Correc and populate them with the set of predicted and corrected atoms, respectively. The precondition for corre requires that an atom is already in the Predic set, and ensures that, after successful completion, the atom is no longer in the Predic set, but is instead in the Correc set.

Atoms belong to molecules, which are handled by the H2O module. A molecule tracks the position and velocity of the three atoms belonging to that molecule; they can be in a variety of conceptual states, indicating not only whether their position has been predicted and corrected but also whether the intra-molecule force corrections have been applied, whether the molecule’s forces have been scaled, etc. We verify the invariant that when the molecule has been corrected, the atoms in the molecule are also corrected. The interface of the H2O ensures that the operations on the molecule may only be invoked in a certain order; for instance, only molecules in the Kineti set (which have had their kinetic energy calculated) may be passed to the bndry procedure.

The ensemble module manages the collection of molecule objects. This module stages the entire simulation by iterating over all molecules and computing their positions and velocity over time. The ensemble module uses boolean predicates to track the state of the computation; when boolean predicate INTERF is true, for example, then the inter-molecule force computation has been carried out on all molecules in the simulation. Our analysis verifies that the boolean predicates, representing program state, satisfy the following ordering relationship:

```
Init ~ INITIA ~ PREDIC ~ INTRAF ~ VIR ~ ...
```

Our specification relies on an implication from boolean
molecule objects, which can be ensured by the array analysis plugin.

Finally, the main module is responsible for initializing the state of the ensemble module and printing out the final state of the system. In the water benchmark, the main loop can only be executed after an initial iteration of the computation has proceeded; our analysis verifies that the appropriate boolean predicate always holds before the main loop is initiated.

The water properties verified by our analysis help ensure that the computation's phases execute in the correct order; they are especially valuable in the maintenance phase of a program's life, when the original designer, if available, has long since forgotten the program's phase ordering constraints. Our analysis' set cardinality constraints also prevent empty sets (and null pointers) from being passed to procedures that expect non-empty sets or non-null pointers.

8.8 Discussion

Most analyses that check safety properties are perceived to be valuable because of the potential they hold for finding (and enabling the elimination of) errors in programs. Our system did identify a number of errors in programs. Furthermore, our use of the system had a profound impact on the development of our benchmark programs. In particular, the need to develop the specifications forced us to think more deeply about the intended structure and behavior of the program. We believe that this process eliminated much ambiguity about the program's behavior before we started developing the implementation sections, reducing the number of coding errors that found their way into these modules.

In general, we found abstract sets to be an appropriate formalism for our specifications. They allowed us to effectively capture, in a natural and easy to use way, many relevant properties of the data structures in our example programs. Of course, this abstraction does not capture all potentially relevant aspects (for example, ordering or mapping relationships between objects inserted into and retrieved from an encapsulated data structure), but it is decidable, natural, and worked well for us in our examples.

One surprise was that our system found substantially more errors in specification sections than in implementation sections. In particular, we sometimes found that our initial specification was not strong enough and we had to add more clauses before the properties were actually true. This process substantially improved our understanding of what the program was actually doing.

In some cases the system surprised us with the sophistication of the properties that it was able to check. Because the modularity of our analysis approach eliminates any need for the analyses to scale to sizable programs, we were able to deploy very powerful analyses that could check quite strong program properties. Very few program analyses, for example, are able to verify that an implementation correctly processes every object in a given data structure. Nevertheless, the extreme precision of our analyses enabled them to check this kind of property in some of our test programs.

9 Related Work

We are aware of no previous research that allows multiple different analyses to analyze different parts of the program and share their results to detect or verify important properties that span parts of the program analyzed by different analyses. We survey related work in shape analysis, typestate systems, boolean algebra decision procedures, and program checking tools in general.

Shape Analysis. The goal of shape analysis is to verify that programs preserve consistency properties of (potentially-recursive) linked data structures. Researchers have developed many shape analyses and the field remains one of the most active areas in program analysis today [23, 30, 31]. These analyses focus on extracting or verifying detailed consistency properties of individual data structures. While these analyses are very precise, the detail of the properties that they must track have limited their scalability. One of our primary research goals is to enable the application of these sophisticated analyses in a modular fashion, with each analysis operating on only that part of the program relevant for the properties that it is designed to verify.

Typestate Systems. Typestate systems track the conceptual states that each object goes through during its lifetime in the computation [12, 34]. They generalize standard type systems in that the typestate of an object may change during the computation. Our approach enables the checking of properties that generalize typestate properties [26]. The developer can simply use sets to model typestates: if an object should be in a given typestate in the typestate system, it is a member of the corresponding set in our system.

Decision Procedures for Boolean Algebras. We use first-order logic formulas in the language of boolean algebras as the basis of our module specification language. The decidability of the satisfiability problem for the first-order theory of boolean algebras dates back to [28, 33] and is presented in [2, Chapter 4]. The complexity of this problem is alternating exponential time [22]. To our knowledge, the only tool that can decide the first-order theory of boolean algebras is the MONA [19]; it implements the more general decision
procedure for monadic second-order logic over trees, and has non-elementary complexity in general but adequate performance in practice for the problems that arise in our program analysis framework. A decision procedure for an extension of boolean algebras with Presburger arithmetic operations is presented in [24].

Program Checking Tools. ESC/Java [6] is a program checking tool whose purpose is to identify common errors in programs using program specifications in a subset of the Java Modelling Language [6]. ESC/Java sacrifices soundness in that it does not model all details of the program heap, but can detect some common programming errors. Other tools focus on verifying properties of concurrent programs [7,29] or device drivers [3,16]. One important difference between this research and our research is that our research is designed not to develop a single new analysis algorithm or technique, but rather to enable the application of multiple analysis that check arbitrarily complicated properties within a single program.

10 Conclusion

The program analysis community has produced many precise analyses that are capable of extracting or verifying quite sophisticated data structure properties. Issues associated with using these analyses include scalability limitations and the diversity of important data structure properties, some of which will inevitably elude any single analysis.

This paper shows how to apply the full range of analyses to programs composed of multiple modules. The key elements of our approach include modules that encapsulate object fields and data structure implementations, specifications based on membership in abstract sets, and invariants that use these sets to express (and enable the verification of) properties that involve multiple data structures in multiple modules analyzed by different analyses. We anticipate that our techniques will enable the productive application of a variety of precise analyses to verify important data structure consistency properties and check important typestate properties in programs built out of multiple modules.

Acknowledgements. We thank Anders Møller for help with the PALE and MONA packages. The second author thanks Rustan Leino for discussions on weakest preconditions, Darko Marinov for discussion on symbolic execution, and Andreas Podelski for discussion on quantifier elimination in program analysis.

References


Verification on the WEB of Name-Passing Process Calculi

Gianluigi Ferrari, Ugo Montanari, and Emilio Tuosto

Dipartimento di Informatica, Università di Pisa

Abstract. Verification toolkits are typically of rather special and, if one compares different related toolkits to each other, complementary functionalities. It has therefore been proposed to integrate different related toolkits within common environments. We argue that Web service technologies can serve as an innovative platform for addressing integration issues. Indeed, service integration in general is certainly an essential ingredient of the Web service paradigm. Furthermore, directory-like techniques will help to tackle integration of verification methodologies.

1 Introduction

In the last couple of years distributed applications over the World-Wide Web, Web for short, such as e-commerce, file sharing, have attained wide popularity, spawning an increasing demand for evolutionary paradigms in designing and controlling them. Uniform mechanisms have been developed for handling computing problems which involve a large number of heterogeneous components that are physically distributed and (inter)operate autonomously.

These developments have begun to coalesce around a paradigm where the Web is exploited as a service distributor. A service in this sense is not a monolithic Web server but rather a component available over the Web that others might use to develop other services. Conceptually, Web services are stand-alone components in the Internet. Each Web service has an interface accessible through standard protocols and, at the same time, describing the interaction capabilities of the service. Applications over the Web are developed by combining and integrating Web services. Moreover, no Web service has pre-existing knowledge of what interactions with other Web services may occur.

We argue here that this paradigm can be fruitfully applied to the problem of integrating formal verification toolkits. We are motivated by the following two observations:

- There is a need for and a profit to be gained from integrating related formal verification tools since despite being related they are often of rather special and, at the same time, at least partly complementary functionality.

* Work supported by European Union project PROFUNDIS, Contract No. IST-2001-33160.
There is a sound conceptual basis for integration since verification toolkits are semantics-based in the sense that they are implementations of well-understood mathematical theories. In principle it is therefore easy to match (part of) the interfaces of related toolkits so that they can work together.

These observations have been made before and indeed other concepts for toolkit integration have been put forward (e.g. [5]). Web services are, however, a new approach to the topic.

Our approach is distributed, thereby moving the issue to the realm of coordination. For two fundamental reasons, this way of dealing with the problem seems to be natural:

1. Some tools are but others are not or only to a limited degree portable or even available for download, and this situation is not likely to change anytime soon, for a variety of reasons. Distribution has some intrinsic advantages over centralised approaches also, such as availability, reconfigurability and openness.

2. Distributed tool integration seems to be best-viewed from coordination standpoint, since we are aiming at what we call deep integration: There shall be automatic means for assigning sub-tasks belonging to any verification run to those node that are most appropriate for solving them. That in turn requires both a platform and a language that allow users to direct where, at different levels of abstraction. This requirement can be regarded as almost synonymous with coordination.

Shallow integration would just mean that users could choose between different tools for tackling any verification problem as a whole.

Our immediate focus is on toolkits for verifying mobile processes (e.g. [10, 11, 17]) in the sense of the π-calculus [14] and related higher-level toolkits for verifying security protocols (e.g. [2, 15]).

One main idea of the work presented here is to make semantic-based verification toolkits available as Web services, using standards such as XML, WSDL and SOAP. Another main idea is to establish directories for publishing such Web services. On this basis, then, we will be able to play out the fact that verification tools are semantics-based, as that characteristic should allow us to provide powerful automated matching facilities for services.

Verification web services plus automated verification web service directories thus provide a platform within our concept for distributed, deeply integrated and therefore coordinated verification. On top of it, there is a verification scripting facility so that users are able to specify how the sub-tasks within any verification run are to be carried out. Beyond the current prototype, we envisage that an important role in that will eventually be played by the trading functionality embedded in semantics-based directories. Specifically, there should be different levels of abstraction in proof scripting, namely goal-oriented, algorithm-oriented, tool-oriented and node-oriented levels.

The rest of this paper presents the prototype of an environment which integrates and coordinates different verification tools via the Web as a service distributor. The development of the verification environment has been performed inside
the Profundis project (see URL http://www.it.um.se/profundis) within the Global Computing Initiative of the European Union. For this reason we called it the Profundis WEB, PWeb for short.

2 Preliminaries

2.1 Web Services

A Web service consists of an interface describing operations accessible by message exchange over the Internet protocol stack. The description of a Web service must cover all details needed to interact with it: the message formats, the transport protocols, and so on. Hence, Web services are a programming technology for distributed systems based on Internet standards. However, Web services are not just another object-based paradigm for distributed systems. Indeed, they promote a service-oriented programming style which is different from the standard user-to-program style [13, 18]. The service oriented programming metaphor is usually characterised in terms of publishing, finding and binding cycle.

To publish-find-bind in an interoperable way Web services rely on a stack of network protocols. The building block of this protocol is the Simple Object Access Protocol (SOAP) [4]. SOAP is an XML-based messaging protocol defining standard mechanism for remote procedure calls. The Web Service Description Language (WSDL) [7] defines the interface and details service interactions. The Universal Description Discovery and Integration (UDDI) protocol supports publication and discovery facilities [19]. Finally, the Business Process Execution Language for Web Services (BPEL4WS) [9] is exploited to produce a Web service by composing other Web services.

2.2 Verification Toolkits for Mobility and Security

Over the years several semantic-based verification toolkits have been designed and experimented to formally address some issues raised by software development. The Concurrency Workbench [8], for example, was developed at the University of Edinburgh and performs analysis on the Calculus for Communicating Systems (CCS). The Mobility Workbench (MWB) [17], developed at the University of Uppsala, does similar analysis but on the \( \pi \)-calculus. The History-Dependent Automata Laboratory (HAL) [10] supports verification of logical formulae expressing properties of the behaviour of \( \pi \)-calculus agents.

Most of the semantic-based verification environments have been developed independently of each other and there is no guarantee that they can interoperate so that the verification of certain properties is the result of a collaboration among the toolkits.

In the verification community the standard approach to deal with the integration issue is to provide a coordination infrastructure based on common format. The FC2 format [3] is an illustrative example of this approach. The FC2 format provides a language to represent automata. An automaton is represented in the
FC2 format by means of a set of tables that keep the information about state names, arc labels, and transition relations between states. FC2 allows interoperability among verification toolkits.

A different approach is exploited by the Electronic Tool Integration Platform (ETI) initiative [5]. ETI is a web-based infrastructure for the interactive experimentation of verification toolkits. The coordination middleware (HLL) provides the "glue" to integrate the different verification toolkits.

The PWeb proposes itself as an experiment to address the integration issue by exploiting Web services. The PWeb prototype implementation has been conceived to support reasoning about the behaviour of systems specified in some dialect of the $\pi$-calculus. The PWeb integrates and coordinate the facilities of some verification toolkits provided as Web services. The MWB and HAL are two of the services of the PWeb. Hereafter, we briefly list the main features of the other services of the PWeb.

TRUST The TRUST toolkit [16,15] relies on an exact symbolic reduction method, combined with several techniques aiming at reducing the number of interleaving that have to be considered. Authentication and secrecy properties are specified in a very natural way, and whenever an error is found an intruder attacking the protocol is given.

MIHDA The MIHDA toolkit [11] performs state minimisation of History-Dependent (HD) automata. HD automata are made out of states and labeled transitions; their peculiarity resides in the fact that states and transitions are equipped with names which are no longer dealt with as syntactic components of labels, but become explicit part of the operational model. This allows one to model explicitly name creation/deallocation, and name extrusion; these are the distinguished mechanisms of name passing calculi. MIHDA has been exploited to perform finite state verification of $\pi$-calculus specifications.

STA STA (Symbolic Trace Analyzer) [2] implements symbolic execution of cryptographic protocols. A successful attack is reported in the form of an execution trace that violates the specified property.

ASPASYA ASPASYA (Automatic Security Protocol Analysis via a SYmbolic model checking Approach) [1] relies on a symbolic technique to model check properties of cryptographic protocols. Security properties are expressed via a logic that predicates over data exchanged in the protocol and observed by an intruder in the execution environment, and also over the "presumed" identities of the protocol principals. ASPASYA allows varying the intruder’s knowledge, the portion of the state space to be explored, and the specification of implicit assumptions that are very frequent in security. The user can opportune mix those three ingredients for checking the correctness of the protocol without modifying neither the protocol specification nor the specification of the desired properties.
3 The PWeb Directory Service

The PWeb implementation has been conceived to support reasoning about the behaviour of systems specified in some dialect of the \( \pi \)-calculus. It supports the dynamic integration of several verification techniques (e.g., standard bisimulation checking and symbolic techniques for cryptographic protocols). The PWeb has been designed by targeting also the goal of extending available verification environments (Mobility Workbench [17], HAL [16]) with new facilities provided as Web services. This has given us the opportunity to verify the effective power of the Web service approach to deal with the reuse and integration of “extending” modules.

The core of the PWeb is a directory service. A PWeb directory service is a component that maps the description of the Web services into the corresponding network addresses. Moreover, it supports the binding of services.

The PWeb directory maintains references to the toolkits it works with. Every toolkit has an end-point in the directory service through the WSDL specification. As expected, the WSDL specification describes the interaction capabilities of the toolkit; namely which methods are available and the types of their inputs and outputs. In other words, the WSDL specification describes what a service can do, how to invoke it and the supported XML types (more precisely the XML Schema definitions XSD).

For instance, the WSDL specification of MIHDA provides the description of the reduce service. The description of the reduce service refers to the XML description of the HD-automaton describing the behaviour of a \( \pi \)-calculus agent. The invocation of this service on a given HD-automata performs the state minimisation of the HD-automata. The WSDL description of the reduce service of the MIHDA toolkit is displayed in Figure 1.

Notice that there is nothing preventing several directory services to connect to the same toolkits, or to include references to other directory services. Hence, the PWeb is basically a peer-to-peer system.

The PWeb directory service has two main facilities. The publish facility is invoked to make available a toolkit as Web service. The query facility, instead, is used to discover which are the services available. The query provides the service discovery mechanism; it yields the list of services that match the parameter (i.e. the XSD type describing the kind of services we are interested in).

The service discovery mechanisms is exploited by the trader engine. The trader engine manipulates pools of services distributed over several PWeb directory services. It can be used to obtain a Web service of a certain type and to bind it inside the application. The trader engine gives to the PWeb directory service the ability of finding and binding at run-time web services without “hard-coding” the name of the web service inside the application code. In other words, the trader engine provides the resource discovery mechanism for PWeb directory services.

The following code describes the implementation of a simple trader for the PWeb directory.
Fig. 1. The MIHDA WSDL-specification
import Trader

offers = Trader.query("reducer")

mihda = offers[0]  # choose the first

offers = Trader.query("model-checking")

hal = find_neighbor(offers)  # choose the service only among neighbors

offers = Trader.query("bisimulation-checking")

mub = offers[0]  # choose the first

The trader engine allows one to hide network details in the service coordination code. A further benefit is given by the possibility of replicating the services and maintaining a standard access modality to the Web services under coordination. For instance, the code

offers = Trader.query("security checker")

can be used to obtain a coordination code that, at run-time, is able to find, bind and finally invoke any service registered as “security checker”. In the PWeb prototype implementation both TRUST and STA are registered as security checkers.

The PWeb directory service is built using Zope [20]. Zope is a (open source) framework for building web applications and is designed to allow administrators to build complex and easily maintainable web servers with a minimum amount of work. Dynamic content is supported through the use of databases which in turn can be updated through web interfaces. Zope is also highly configurable and fully object oriented. New objects can be added and inherited from if the need arises allowing for existing features to be tailored to user needs. There is also a robust security system which allows administrators to manage user privileges. One of the main advantages of Zope is that it is portable. It runs on a variety of machines infrastructures including Windows 2000/NT/XP, Linux, Solaris and Mac OS X.

As a final remark we want to point out that the trader engine provides facilities which are similar to the CORBA trader. The CORBA trader is used to query object infrastructures for specific applications and components.

The current prototype implementation of the PWeb directory service can be exercised on-line at the URL http://jordie.di.unipi.it:8080/pweb.

3.1 Service Coordination

The fundamental technique which enables the dynamic integration of services is the separation between the service facilities (what the service provides) and the mechanisms that coordinate the way services interact (service coordination). In our experiment, the service coordination language is PYTHON. PYTHON is an
interpreted object oriented scripting language which is widely used to connect existing components together.

An example of service coordination is illustrated in Figure 2 to verify a property of a specification, i.e. to test whether a π-calculus process $A$ is a model for a formula $F$.

: try:

    hd = mihda.compile( A )

    reduced_hd = mihda.reduce( hd )

    reduced_hd_fc2 = mihda.Tofc2( reduced_hd )

    aut = hal.unfold( reduced_hd_fc2 )

    if hal.check( aut, F ):
        print 'ok'
    else:
        print 'ko'

except Exception, e:
    print "*** error ***"

Fig. 2. Coordinating HAL and Mihda services

We can briefly comment on the orchestration code of Figure 2. Variables mihda and hal, have been linked by the trader engine to the required services. Now, a service of Mihda is invoked. More precisely, the result of executing the service compile is stored in the variable hd.

Next, hd is minimized, by invoking the service reduce of Mihda; and, by applying the Mihda service Tofc2, the minimal automaton is transformed into the FC2 format. Variable reduced_hd_fc2 contains a HD-automaton in a format suitable for being processes by the HAL service unfold that generate an ordinary automaton from a HD-automaton represented in FC2 format.

Finally, a message on the standard output is printed. The message depends on whether π-calculus process $A$ satisfies the formula $F$ or not. This is obtained by invoking the HAL model checking facility check.

3.2 Querying XML Types

The PWeb directory service includes a database of XML types which keeps track of the relationships among the XSD types which can be exploited as arguments of messages. Whenever a new XSD type is added to the PWeb directory service (e.g
as a result of a service registration), it is compared with the existing XSD types and its relationships with the other registered types are stored in the database. Whenever an application performs a query, the trader engine will provide a list of types which are compatible with the type of the query. In the prototype implementation, this is obtained by a simple script code written in Python.

We are currently investigating more expressive and powerful mechanisms for querying XML types. In particular, we started some experiments in using programming languages specifically designed to manipulate and querying XML data [12, 6].

4 Lessons Learned

We started our experiment with the goal of understanding whether the Web service metaphor could be effectively exploited to integrate in a distributed and coordinated fashion semantics-based verification toolkits. In this respect, the prototype implementation of the PWeb is a significant example.

Our approach adopts a service orchestration model whose main advantage resides in reducing the impact of network dependencies and of dynamic addition/removal of Web services by the well-identified notions of directory of services and trader engine. To the best of our knowledge, this is the first verification environment that specifically addresses the problem of exploiting Web services.

The service orchestration mechanisms presented in this paper, however, have some disadvantages. In particular, they do not exploit the full expressive power of SOAP to handle types and signatures. For instance, the so called “version consistency” problem (namely the client program can work with one version of the service and not with others) can be solved by types and signatures.

References