Micromodels of Software

Daniel Jackson
MIT Lab for Computer Science
200 Technology Square
Cambridge, Mass. 02139
617 258 8471 (voice)
617 258 8607 (fax)
dnj@mit.edu
http://sdg.lcs.mit.edu/~dnj

A micromodel of a program is a tiny, abstract model that captures some crucial aspect of its functionality. To improve the reliability of software, implementations are developed hand-in-hand with micromodels. The code is then checked for conformance with all the relevant micromodels. Micromodels can also themselves be simulated and checked for internal consistency.

Micromodels of Software

Daniel Jackson
MIT Lab for Computer Science
200 Technology Square
Cambridge, Mass. 02139

1 Introduction

The Software Design Group at MIT has been developing an approach to developing reliable software based on *micromodels*: tiny models that capture tricky aspects of a system's design. Micromodelling is a 'lightweight formalism'. But unlike most lightweight formalisms, such as type checking or static analysis [8], micromodelling is non-uniform, and vertical rather than horizontal.

A uniform approach requires that every part of the system receive the same degree of attention, for example, by annotating all procedures with partial specifications. The developer spreads the formalism 'horizontally' over the entire system, in a layer of even depth. The benefit of this approach is that you can then establish basic properties of the system as a whole: 'small theorems about big systems', as Bill Scherlis once put it. Uniform approaches are very valuable, because they can eliminate many of the small errors that plague software developments: null pointers, data races, array bounds violations, and so on.

But it seems unlikely that uniform approaches will scale to more crucial, domain-specific properties. Even if the analysis scale, it's unlikely that developers will be willing to spread a much thicker layer. What's needed is the ability to focus on some crucial aspect of the system, and to check the implementation by devoting significant efforts to small areas, and almost no effort to the rest. The formalism is thus applied vertically, in the critical areas of the code, at multiple levels of abstraction.

In a sense, this is what developers do already. When faced with checking some crucial aspect of a system, a developer will first try and develop a model (often in her mind) of the required behaviour; will isolate the parts of the code that are relevant to that aspect; and will then examine them closely to see if they conform. Our research agenda can be seen as an attempt to support this approach with flexible and powerful tools.

2 Elements

What's needed to make this work?

- A *modelling language* that allows models of crucial aspects to be expressed succinctly and naturally;
- A *simulation tool* for playing with the model, to check that it expresses the intended properties, and has the intended consequences;
- An *extraction tool* for extracting the parts of an implementation that are relevant to a particular micromodel;
- A *checking tool* that checks the extracted code fragments against the model.

We've made some progress on each of these fronts. A new version of our modelling language, Alloy, is more flexible and powerful than its predecessor [4]. It has generic types, relations of arbitrary arity, integers, and higher-order quantifiers.

Alloy's tool can perform simulations and check intended properties of a model using a strategy based on translation to SAT [3]. As SAT solvers get faster and faster, our tool becomes more powerful. We're about to incorporate Chaff [9], a new solver that preliminary experiments suggest will improve performance of our tool by an order of magnitude.

We have shown the feasibility of checking properties of code using the same tool, by translating code directly into Alloy [2], and have also begun to investigate expressing tricky aspects of object-oriented code – such as views, as created by methods in the Java API such as *iterator*, *sublist* and *keySet* – in Alloy [6].

Finally, we've started work on the extraction process. A key feature of our approach is that code will be extracted into a model that is intelligible to the developer: not a skeletal program in some low-level intermediate language suitable only for analysis. We're looking at using message sequence charts as the organizing principle for identifying and extracting parts of the code relevant to particular interactions. A preliminary example is given in [6].

3 Rationale

Why is the problem of checking that a design captures the right properties, and that an implementation conforms to it, important? The need for more reliable software is clear: more dependence on software for critical infrastructure; more embedded and pervasive software, deployed in more varied and unanticipated environments; the unpalatable cost of testing; the threat of consumer rebellion against shoddy software; and so on.

Why models? Programming languages seem to have run their course. Java embodies many of the best ideas of the last few decades, and it's not clear that there will be any way of expressing the *details* of a desired computation more absractly and succinctly. Models bring not just abstraction but *partiality* – the ability to focus on some aspect of the system behaviour without describing all the details, and completely ignoring other aspects.

Why analysis rather than synthesis? Our approach has something in common with aspect-oriented programming [7], in recognizing cross-cutting concerns, some of which are more crucial and deserving of attention than others. Perhaps an implementation could be generated automatically from our models, putting them together like aspects. This seems unlikely, though; our models are much more partial than aspects, and rely for their succinctness on declarative features that are not easily translated into code.

Why our particular modelling and analysis approach? Our language is essentially first order logic, with sets and relations, and some structuring. Despite the prominence of first-order logic in our view of mathematics, it's been used surprisingly little in program analysis (but see [10]). Perhaps the reason is that a logic with quantifiers (and not just Horn clauses) has been assumed to be too intractable for practical use. This might have been true 20 years ago, but Moore's Law, and progress in SAT solvers, have changed the situation dramatically.

4 A Challenge Problem

In exploring the redesign of an air-traffic control component [1], we built a prototype replacement of a component. This component, the *Communications Manager* of CTAS, is essentially a packet forwarding program. We implemented it in an action-machine idiom: a central registry holds a (dynamically adjustable) association of packet types with actions. When a packet is received, actions associated with its type are executed.

For a program using this idiom, we might want to establish certain properties of the action registration structure: for example, that the actions for a packet type do not affect the registrations of the type itself, or that actions can be executed in any order (at least with respect to their effect on registrations). These properties are easy to express in a micromodel, but hard to see directly in the code.

We are investigating how to extract relevant code from a program that uses this action-machine idiom to check that the idiom is used in accordance with these desired properties. We've written another, simpler program, that uses the same idiom – a text processing program used in the preparation of this paper – and shows the same compli-

cations, but in a more limited and manageable setting. A design overview of this program – called *Tagger* - along with its source code is available online [5], and some of the properties are described in [6].

Acknowledgments

The research program described here is being funded by ITR grant #0086154, *Design Conformance*, from the National Science Foundation, whose investigators are Daniel Jackson & Martin Rinard (MIT computer science), John Hansman (MIT Aero/Astro) and David Schmidt (Kansas State computer science), by a grant from NASA, and by an endowment from Doug and Pat Ross.

References

[1] Daniel Jackson & John Chapin. Simplifying Air-Traffic Control: A Case Study in Software Design. *IEEE Software*, May/June 2000, pp. 63–70. Available at: http://sdg.lcs.mit.edu/~dnj/publications.

[2] Daniel Jackson & Mandana Vaziri. Finding Bugs with a Constraint Solver. *International Symposium on Software Testing and Analysis*, Portland, Oregon, August 2000. Available at: http://sdg.lcs.mit.edu/~dnj/publications.

[3] Daniel Jackson. Automating first-order relational logic. *ACM SIGSOFT Conference on Foundations of Software Engineering*, San Diego, California, November 2000. Available at: http://sdg.lcs.mit.edu/~dnj/publications.

[4] Daniel Jackson, Ilya Shlyakhter and Manu Sridharan. A Micromodularity Mechanism. *ACM SIGSOFT Conference on Foundations of Software Engineering / European Software Engineering Conference*, Vienna, Austria, September 2001. Available at: http://sdg.lcs.mit.edu/~dnj/publications.

[5] Daniel Jackson. 6170: Laboratory in Software Engineering; Lecture 18: Case Study, Tagger; October 2001. Available at: http://web.mit.edu/6.170/www/lectures/.

[6] Daniel Jackson & Alan Fekete. Lightweight Analysis of Object Interactions. *Conf. on Theorectical Aspects of Computer Science* (TACS'01), Sendai, Japan, October 2001. Springer-Verlag, LNCS 2215, pp. 492-513. Available at: http://sdg.lcs.mit.edu/~dnj/publications.

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[8] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. *ESC/Java User's Manual*. Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[9] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an Efficient SAT Solver. *39th Design Automation Conference*, Las Vegas, June 2001.

[10] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, Jan. 20–22, 1999, ACM, New York, NY, 1999.