Automated Development and Run-time Adaptation of Interactive Distributed Applications

(White Paper)

B. Cheng, L. Dillon, K. Stirewalt, P. McKinley, S. Kulkarni, and J. Lee

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
(http://www.cse.msu.edu/sens)

October 2001

Abstract

In order to provide focus in discussing new techniques and new paradigms for software development, we consider a class of applications that is gaining momentum in terms of widespread use, while exhibiting safety-critical properties that require rigorous and formal development techniques. *Interactive distributed applications (IDAs)* are those that involve direct interaction with users and whose processing and data components are distributed across a network. IDAs exhibit characteristics, such as interactivity, concurrency, non-determinism, heterogeneity, and dynamic adaptation, that complicate both their development and reasoning about their behaviors. To facilitate develop of quality IDAs various aspects of their construction and maintenance must be automated. This white paper proposes three main thrust areas that enable interactive distributed applications to be developed and maintained for fast-changing requirements and environments: integrating informal and formal techniques, support for end-to-end development, and support for run-time adaptation of software.

1 Overview

In order to provide focus in discussing new techniques and new paradigms for software development, we consider a class of applications that is gaining momentum in terms of widespread use, while exhibiting safety-critical properties that require rigorous and formal development techniques. *Interactive distributed applications (IDAs)* are those that involve direct interaction with users and whose processing and data components are distributed across a network. Examples of this increasingly important class of applications include distributed data management systems, on-board driver/pilot navigation assistance systems, computer-supported cooperative work, Web-based distance education, command and control, and a variety of public safety services. A requirement of these systems is that they must rapidly incorporate new functionality and accommodate new hardware technologies to forestall becoming obsolete. To develop quality software under these circumstances, various aspects of IDA construction and maintenance must be automated. The safety-critical nature of these applications and the need for automation motivate the need for formal methods. This white paper describes three complementary thrust areas to be addressed in order to rigorously develop IDAs and provide support for run-time adaptation: integrating informal and formal techniques, support for end-to-end development, and support for run-time adaptation of software.

Software development is inherently difficult to automate. IDAs exhibit three characteristics that exacerbate this difficulty. First, IDAs exploit a combination of new technologies for which rigorous engineering models are not yet available. Examples include network services for streaming real-time data across wireless networks, middleware and software agent technologies, and language facilities that support secure mobile code. Without rigorous models, an automated code generator cannot produce programs that correctly use these features, and an automated systems design and analysis tool cannot predict the effect of these features on a system under design. Second, IDAs interact with other independent systems, including human users. These interactions must be designed explicitly. By nature, this task involves extensive prototyping and testing, which are difficult to fully automate. Third, IDAs comprise multiple levels of concurrent, communicating components executing in a distributed environment. Programs that operate in such an environment are subject to race conditions, scheduling anomalies, and dynamic environment conditions. Automated program analyzers and code generators must deal with this complexity in order to correctly model system behavior and produce programs that operate correctly.

Despite these obstacles, we believe that fundamental advances toward end-to-end automation of IDA development can be made if each of the following issues are addressed concurrently. First, the integration of different tools must be structured around explicit program representations with formally defined semantics. Such formal representations are needed to support behavioral analysis and to enable tracing of design requirements from high-level models to low-level code. Second, since programmers usually find formal specifications of program behavior difficult to use directly, these formal representations must be derivable from graphical modeling techniques with which programmers are already comfortable, and the formal analysis techniques must be packaged into easy-to-use tools that consume and produce such graphical models. Third, automated techniques must support the construction of software that is adaptable at runtime to the changing environment. This latter property is increasingly important as computing devices become smaller and more mobile.

The remainder of this white paper is organized as follows. Section 2 describes issues with integrating informal and formal approaches to software development. Section 3 discusses end-to-end automation tradeoffs and Section 4 discusses run-time adaptation of software. Finally, in Section 5, we briefly describe two projects in the Software Engineering Laboratory at Michigan State University that helped shape our thoughts on issues discussed in this white paper.

2 Using Both Informal and Formal Methods

Recently, many researchers in the software engineering community are looking at integrating informal and formal techniques [1, 2, 3, 4]. Like these researchers, we believe techniques that capitalize on both informal and formal methods hold great promise for producing fundamental advances in development and maintenance

of complex software systems. Informal methods are typically graphical in nature and easy to use, but it can also be easy for the developer to construct erroneous models since there is no way to rigorously check the diagrams. In contrast, formal techniques are well-defined and are amenable to automated analysis, such as simulation, syntax and semantics checking, model checking, rewriting techniques, and theorem proving. Unfortunately, some formal methods may be difficult to use because the formal specifications may not be easy to construct from scratch. Moreover, modifications to the specifications may be difficult to effect.

We contend that integration of informal techniques and formal techniques can largely overcome the disadvantages to each approach and highlight the advantages. Formal specification techniques can enhance the application of informal techniques, by providing the means for rigorously analyzing properties that they informally represent. Rigorous analysis can uncover errors that are difficult to spot in reviews of informal models, which would otherwise allow errors to propagate to the later stages of development thereby increasing the overall development costs. The analysis tools for the formal techniques enable developers to perform numerous types of analysis on the informal diagrams that would otherwise not be possible. Examples include intra- and inter-model consistency checking, specification verification and validation through the use of model checking and simulation techniques, behavior simulation, and rapid prototype development. The formal techniques enable developers to explore rigorous and automated design refinement techniques, including code generation and test case generation. Formal techniques can reduce the burden currently placed on testing.

We further believe that the benefits of integrating informal and formal techniques are maximized by including visual support for formal analysis tools that relate back to the informal notation. Specifically, visualization techniques can be used to interpret the analysis results from formal analysis tools in order to indicate the source of errors in the original diagrams and convey the errors in alternative, easy to understand formats. Visualizations enable the intricacies of formal specifications to be largely "hidden" from the developer, thus allowing for the construction and refinement of system specifications at the informal level, while taking advantage of the rigorous analysis capabilities of formal methods.

3 End-To-End Engineering Tradeoffs

Most of the existing research on automating software development focuses on individual steps in the software engineering life cycle, such as requirements engineering, design processing, code generation, and so on. An important area that has received much less attention is the investigation of *end-to-end* engineering of critical applications. Here, end-to-end refers to the sequence of phases in system development, from early requirements analysis and specification down through implementation, testing, and maintenance. We contend that there is a substantial class of end-to-end engineering tradeoffs that are poorly understood and that are best studied in "vertical" research projects, which span the full software life cycle and involve multiple investigators with diverse software-engineering backgrounds (e.g., in object orientation, distributed systems, fault tolerance, programming languages, compilers, code generation, testing, etc.).

Requirements verification, for example, is an end-to-end engineering problem whose tradeoffs are poorly understood. A requirement might be verified by one of several means. For example, a developer might structure the design and the implementation so that the requirement can be traced through the entire development process. Alternatively, a developer might phrase the requirement as a specification in a formal language, such as LTL or CTL, and then verify this requirement against a model of the system using model checking. Finally, a developer might craft a set of test cases that check adherence to the requirement and apply these test cases to the completed application. Obviously, there are many variables in this problem. The proper approach to use depends on the nature of the requirement, how it is specified, and other factors, such as the composition features available in the implementation language. As systems become increasingly complex and integrated, end-to-end development issues become ever more critical.

Within the general problem of better understanding end-to-end issues and tradeoffs, we believe there are two key challenges. First is how best to capitalize on an enormous and continually growing body of reusable assets, such as class libraries, reference architectures, and UML models that capture the requirements of a

"similar" system. Reuse is an end-to-end problem in itself; however, it clearly affects and constrains other issues, such as requirements verification.

Second is how to build up the tool infrastructure to support designers in balancing end-to-end concerns and making tradeoffs. Conceptually, the development of a system can be viewed as a series of transformations, each of which operates over an artifact in some input language and produces an artifact in some output language, which is usually different (and more concrete) than the input language. Each transformation involves analyzing the input artifact and generating the output artifact, and a multiplicity of different representation languages seems inherent to this problem. In fact, we expect much of the progress in end-to-end engineering methodologies to incorporate key tradeoffs by tailoring these intermediate languages and the corresponding analysis and generation techniques. One promising approach to enable such tailoring is to encapsulate analysis and generation capability into *lightweight components* that compose seamlessly and efficiently within the architecture of larger software-development tool [5]. Here the term lightweight refers to two useful properties. First, analysis capabilities are parameterized by representations of the structures to be analyzed; such parameterization enables assembly of analyzers that perform analysis on the same in-memory representation of a specification, without translating the representation into another form or invoking an external tool. Second, the analysis software is designed for *extension* and *contraction*—in the sense suggested by Parnas [6]—so that the software can be easily tailored to a particular analysis need during assembly.

4 Run-Time Adaptation

The third major area where we believe formal methods will play a larger role in the future is in supporting run-time adaptation and reconfiguration of software. The growth of the Internet and the increased presence of mobile code has led to important early contributions in related areas, for example, proof-carrying code [7] and security models for active networks [8]. However, as computers increasingly pervade many aspects of our daily lives and support various infrastructures upon which we rely, there is a need for software to adapt its behavior continually to changes in that environment. The need for adaptation arises partly from the emergence of a very dynamic and heterogeneous mobile computing infrastructure, and partly from the increasing dependence on distributed software for many types of services (financial, defense, and medical) for which disruption has serious consequences.

The research and development communities have addressed some of the issues raised by adaptation in *middleware*, which executes between the application code and the transport services offered by the network. An adaptive middleware platform can potentially insulate application components from platform variations and changes in network conditions, can support hand off of applications among devices, and can simplify the maintenance of security and fault-tolerance invariants. However, many approaches to adaptive middleware involve *ad hoc* techniques that do not allow multiple dimensions of adaptability to be addressed in a unified manner. Whereas a given external event can potentially affect several different parts of the middleware (and indeed some higher-level functions of an application), such cross-cutting aspects are not expressible in an *ad hoc* framework. Even those middleware projects that take advantage of "principled" approaches, such as computational reflection [9], focus primarily on the mechanisms that enable the code to modify its behavior, rather than on issues of consistency and correctness of the resulting programs.

We contend that formal methods can help to support the design and operation of adaptive software (middleware, in particular) in three important ways. First, development of a unified model of adaptive components may require a shift in programming paradigm that emphasizes encapsulation, rather than inheritance. However, the resulting freedom in delaying and re-associating of bindings requires automation of run-time checking of both functional and nonfunctional properties of the system. Second, *computational reflection* is a powerful mechanism that enables software to observe its behavior (introspection) and change its behavior (intercession). However, a completely open implementation implies that an application can be recomposed entirely at run time, potentially altering the functional behavior of the program. We propose that the use of reflection be coupled with generative design techniques [10, 11] in order to properly support dynamic composition and automated reuse of components. Finally, an adaptive framework should enable

the application programmer to declaratively specify security, fault-tolerance, and QoS preferences and to design applications that adapt to conditions affecting the middleware's ability to satisfy these preferences. To achieve this goal requires the development of a specification language that can define adaptive behavior and constraints in multiple dimensions, as well as a compiler to generate corresponding application-level stubs that make use of adaptive middleware services.

We believe that using formal methods to support middleware and other types of adaptive software can potentially lead to major improvements in the quality and capabilities of many distributed systems. First, a unified model for adaptive components will help application developers evolve existing applications to accommodate new technologies in a systematic manner. Second, raising the level of abstraction in the design of middleware services will enable developers to more easily incorporate application-specific policies and preferences into the middleware and, by insulating the programmer from the details of adaptive middleware, will speed development and testing. Third, tools that support the definition, composition, and reuse of middleware components will enable developers to assess software safety and correctness issues when components are modified and interconnected during software evolution.

5 SENS Projects

Our thoughts on these issues have been shaped, in part, by two projects in the Software Engineering and Networking Systems Laboratory at Michigan State University. First, MERIDIAN (sponsored by NSF Experimental Partnerships Program) is investigating how to use automated software engineering techniques to decrease the time required to develop and maintain IDAs, without sacrificing quality. To validate MERIDIAN, we are conducting case studies that instantiate this tool suite over different types of IDAs, using industry-supplied applications from Motorola, Texas Instruments, NASA/JPL, and Siemens Automotive. Second, RAPIDWARE (sponsored by ONR's Critical Infrastructure Protection Program) is developing new techniques for reusable and dependable middleware that includes runtime support for adaptation to the executing environment. This project is also being conducted with industrial partners, including Cisco, Lucent, and Motorola. Additional information on research projects being conducted in the Software Engineering and Network Systems Laboratory can be found at `http://www.cse.msu.edu/sens`.

References

[1] R. H. Bourdeau and B. H. C. Cheng, "A formal semantics of object models," *IEEE Trans. on Software Engineering*, vol. 21, pp. 799–821, October 1995.

[2] W. E. McUmber and B. H. C. Cheng, "A general framework for formalizing UML with formal languages," in *Proc. IEEE International Conf. on Software Engineering (ICSE01)*, (Toronto, Canada), May 2001.

[3] J. Lilius and I. P. Paltor, "vUML: a tool for verifying UML models," in *Proc. IEEE International Conf. on Automated Software En gineering*, (Cocoa Beach, FL), October 1999.

[4] P. Bose, "Automated translation of UML models of architecture for verif ication and simulation using SPIN," in *Proc. IEEE International Conf. on Automated Software En gineering*, (Cocoa Beach, FL), October 1999.

[5] R. E. K. Stirewalt and L. K. Dillon, "A component-based approach to building formal analysis tools," in *Proc. 2001 International Conf. on Software Engineering (ICSE'2001)*, 2001.

[6] D. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. of Software Engineering*, vol. 5, no. 2, 1979.

[7] G. C. Necula, "Proof-carrying code," in *Proc. 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Langauges (POPL '97)*, (Paris), pp. 106–119, Jan. 1997.

[8] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, 1997.

[9] P. Maes, "Concepts and experiments in computational reflection," 1987.

[10] D. Batory and S. O'Malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Trans. on Software Engineering and Methodology*, vol. 1, pp. 355–398, October 1992.

[11] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.