Towards Re-engineering Legacy Systems for Assured Dynamic Adaptation *

Ji Zhang and Betty H.C. Cheng†
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
{zhangji9,chengb}@cse.msu.edu

Abstract

Increasingly, software must adapt its behavior in response to changes in the supporting computing, communication infrastructure, and in the surrounding physical environment. Since most existing software was not designed to adapt, research on techniques to make legacy software dynamically adaptive has gained increasing interest. Assurance is crucial for adaptive software to fulfill its intended purpose. Correctness is even more critical if it is to be applied in high assurance systems. This paper proposes a model-driven approach to introduce dynamic adaptation to non-adaptive legacy systems while maintaining assurance properties. An aspect-oriented technique is applied to achieve separation of concerns in the implementation.

1 Introduction

Increasingly, software must adapt its behavior in response to changes in the supporting computing, communication infrastructure, and in the surrounding physical environment [1]. Since most existing software was not designed to adapt, research on techniques to make legacy software dynamically adaptive has gained increasing interest. Assurance is crucial for adaptive software to fulfill its intended purpose, namely hardening security, upgrading services, etc. Correctness is even more critical if the software is to be applied in high assurance systems, such as command and control, critical infrastructure protection systems, etc. In these systems, adaptive software development must be grounded upon formalism and rigorous software engineering methodology, which provide high assurance in software. In this paper, we propose a technique to re-engineer legacy software systems in order to make them dynamically adaptive with an emphasis on the assurance of adaptation.

In recent years, two main areas of research for dynamically adaptive software have been pursued. First, techniques have been proposed for various software system layers to enable dynamic adaptation in previously non-adaptive software (e.g., transparent shaping [2], Aura [3]). Second, many researchers have investigated assurance issues for adaptive systems [4], such as the modeling and analysis of abstract behavioral models and architectures. There is a gap between these two research thrusts for adaptive systems: On the one hand, techniques addressing adaptation in legacy code heavily rely on developers' experience and common sense rather than leveraging rigorous verification techniques, such as model checking. On the other hand, techniques addressing correctness in dynamic adaptation using rigorous software engineering techniques focus on abstract models and do not take the models to their implementations.

This paper proposes a model-driven, aspect-oriented approach to introduce dynamic adaptation to legacy systems, while maintaining assurance properties. Our key insight is that UML models, with formally defined semantics, can be used as an intermediate representation to bridge the gap between the formal models used for adaptive software verification and adaptive software implementations. Adaptation designs can be performed on the UML models by creating adaptation UML models that can be automatically translated into formal models using existing tools [5] for formal analysis. We introduce a *cascade adaptation mechanism* for implementing adaptation designs in the programming language for the legacy code. An aspect-oriented technique is applied to the source code of the system to achieve separa-

*This work has been supported in part by NSF grants EIA-0000433, EIA-0130724, ITR-0313142, CCR-9901017, CCF-0541131, and CNS-0551622, and the Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, and a Michigan State University Quality Fund Concept Grant.

†Contact author.

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

tion of concerns in the implementation.

Our proposed approach has the following characteristics. (1) The approach handles existing non-adaptive legacy code, as well as new adaptive programs. (2)The approach provides assurance to the adaptive program as a prerequisite for the program to be used in critical systems. (3) The code for adaptation is separated from the non-adaptive legacy code. The approach is non-invasive to the original legacy source code, i.e., the source code for the system is not altered directly. Non-invasiveness to the source code is important in order to allow the adaptation code and the legacy code to be maintained separately. (4) The points for adaptation are flexible. Many adaptation techniques [6, 2, 7] require the points for adaptation to be separated from the code segment changed by the adaptation. We find that this constraint may impose unacceptable performance penalties in some adaptation scenarios. Thus, we designed our approach to be more flexible, i.e., allowing adaptation to occur at any identifiable point in the program. (5) The modeling technique is capable of expressing the transformation of state information from the source program (i.e. the program from which the system adapts) to the target program (i.e. the program to which the system should adapt).

We have applied our approach to adaptive mobile computing applications, including the development of an adaptive Java pipeline program [8]. Two non-adaptive versions of this program had been previously developed in our lab. However, the development of the adaptive version of the program had been a daunting task because of the algorithmic complexity involved. Our approach reduced developers' burden by separating different concerns and using automated formal analysis tools. The remainder of this paper is organized as follows. Section 2 gives background on earlier work for an aspect-oriented adaptation enabling technique [6], a model-based adaptive software development process [9], and a metamodel-based UML formalization technique [5]. In Section 3, we overview our approach. Section 4 concludes this paper and briefly discusses our future directions.

2 Background

In this section, we overview three techniques that will be extensively leveraged for our proposed approach.

2.1 Aspect-Oriented Adaptation

Yang *et al* [6] previously developed a two-phased AOP-based technique to enable adaptation in legacy software using AspectJ. In the first phase, occurring at compile time, an aspect fragment, called *behavior adaptor*, defines the points in a legacy program at which "traps" need to be inserted. An AOP compiler, such as the AspectJ compiler, weaves the behavior adaptor into the legacy code to make the legacy code "adapt-ready", i.e., capable of changing behavior. During the second phase, at runtime, an adaptation kernel, i.e., a loose federation of concern-specific adaptation managers, checks execution conditions of the software and performs appropriate adaptive actions according to a dynamically reloadable *rule base*. By using AOP techniques, their approach fully separates the application code (non-adaptive) from the dynamic adaptation concerns.

2.2 MASD: Model-Based Adaptation

A number of techniques (e.g. process algebra-based approaches) [10, 11, 12] focus on verifying that an adaptive program is correct in terms of satisfying a given set of properties. We recently proposed a top-down, model-based adaptive software development process (MASD) [9] to provide assurance in adaptive software. In this process, we start from the high-level goal for the adaptive software. By applying goal-driven requirements analysis [13, 14], we derive a set of execution domains and the local properties for each domain, and a set of global invariants. The local properties for a domain are the properties the adaptive software must exhibit while executing in the domain. Global invariants are the properties the adaptive software must satisfy throughout its execution. We create a formal Petri net model for each domain and verify the model against the local properties for the domain. For each adaptation from a source model to a target model, we create intermediate states and transitions connecting the source model to the target model. The quiescent states for adaptation (the states from which the adaptation may start) are identified by the transitions emitting from the source model. The source model, the target model, and the adaptive states and transitions form an adaptation model. The adaptation Petri net models are then verified against global invariants. Finally, the Petri net models are leveraged in the generation of rapid prototypes and in model-based testing. The MASD technique focuses on the development of new adaptive software, rather than legacy code. The support for generating executable code is only for the requirements level (as prototypes), instead of the production-level code.

2.3 UML Formalization Technique

McUmber and Cheng developed a general framework [5] based on mappings between metamodels (class diagrams depicting abstract syntax) for formalizing a subset of UML diagrams in terms of different formal languages, including Promela [5]. The formal (target) language chosen should reflect and support the intended semantics for a given domain (e.g., mobile computing systems). This formalization framework enables the construction of a set of rules

for transforming UML models into specifications in a formal language. The resulting specifications derived from UML diagrams enable either execution through simulation or analysis through model checking, using existing tools. The mapping process from UML to a target language has been automated in a tool called Hydra [5]. We use Hydra to enable the formal analysis of the UML diagrams created for the adaptive systems. We also use the Hydra framework as the basis for reverse engineering Java programs to obtain behavioral models.

3 Bridging the Gap

In this paper we show how the aspect-oriented adaptation enabling technique [6], the MASD process, and the metamodel- based formalization technique can be integrated and leveraged to bridge the gap between formal models for adaptive systems and adaptation implementations. Although our discussion focuses on legacy code, we also show how our approach, with minor changes, can be applied to newly developed code as well.

Our approach addresses three phases in the development of adaptive software: the requirements analysis phase, the model design and analysis phase, and the implementation phase. As shown in Figure 1, the overall process includes the following four steps: Step (1) occurs in the requirements analysis phase. We perform requirements analysis to select a set of non-adaptive legacy programs $P_1, P_2, \cdots P_k$ (each of which differs from others by one or more segments of code), and a set of properties that must be satisfied by the programs. We specify two types of properties: *Local properties* specify what must be satisfied by each legacy program individually. *Global invariants* specify properties must be satisfied by the adaptive program throughout its execution, regardless of adaptations. Step (2) occurs in the model design and analysis phase. The programs are translated into UML Statechart diagrams $M_1, M_2, \cdots M_k$, (named *steady-state models*). These models will be translated into formal models and verified against their local properties using automated tools [5]. Step (3) also occurs in the model design and analysis phase. After the steady-state models are verified, developers must create an *adaptation model* $A_{i,j}$, also in terms of UML Statechart diagrams, for each adaptation from program P_i to P_j. The adaptation models are then translated to formal models and verified using formal analysis tools against global invariants. Step (4) occurs in the implementation phase. After the adaptation models are verified, they are then integrated and translated into adaptive programs. Our approach addresses the following questions in this step: What mechanism do we use to make legacy code adaptive? Where and what code should be inserted in the legacy code so that the implementation faithfully reflects the adaptation design?

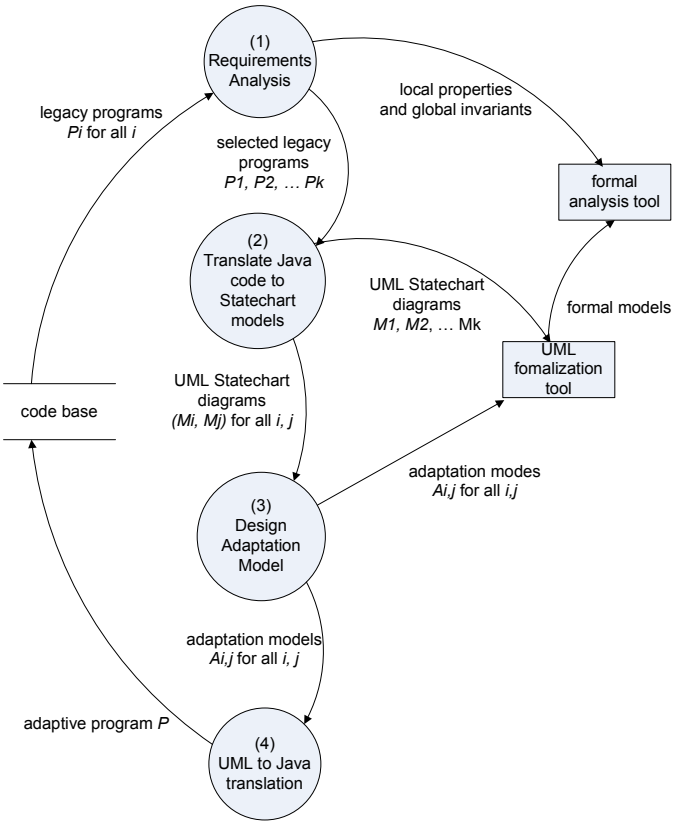

Figure 1. The dataflow diagram for the proposed approach

We next describe each phase in more detail. For discussion purposes, we focus on the adaptation of one adaptive component from one source behavior to one target behavior. Due to space constraints, we omit the description of this technique extended to support collaborative adaptation [15].

3.1 Adaptation Requirements Analysis

We use a goal-based approach [16, 13] to analyze the requirements for adaptive software [9]. As shown in Figure 2, an adaptive program is intended to achieve a high-level goal under different run-time environmental conditions. We first determine the set of environmental conditions (formally defined as domains) $D_1, D_2, \cdots D_k$, in which the adaptive program is required to execute. The requirements for these domains $R_1, R_2, \cdots R_k$, respectively, are those properties that enable the software to achieve the high-level goal in the corresponding domains. We specify two types of requirements: global invariants and local properties. Global invariants usually specify those properties common to all the domains and need to be preserved throughout the execution of the program. Local properties are those specific

to each execution domain. In order to facilitate formal verification of the requirements, we express these requirements in Linear Temporal Logic (LTL). Assume that we have a legacy code base comprising a set of non-adaptive legacy programs and the properties associated with each program. After the set of requirements are specified, we query the code base using the local properties and select the set of non-adaptive legacy programs $P_1, P_2, \cdots P_k$ to be used in the domains $D_1, D_2, \cdots D_k$, respectively. In this section, we assume that the legacy code for all the requirements already exists.

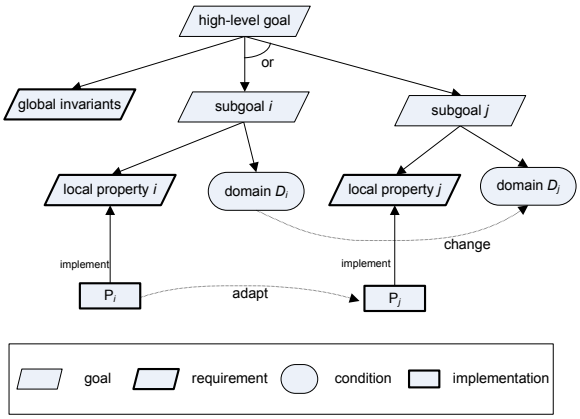

Figure 2. Goal model for adaptive software

Next, we analyze how the execution domains of the program may change at run time, and how the adaptive program should respond. Consider the case where the program is initially running P_i in domain D_i. A change of domain from D_i to D_j may warrant an adaptation from P_i to P_j depending on the cost to develop such an adaptation and the overhead that may be incurred during the adaptation.

3.2 Adaptation Design and Analysis

After selecting the set of legacy programs and adaptations, we create adaptation models in UML using the following steps: First, we reverse engineer each legacy program P_i to generate a UML model (Statechart diagram) M_i. Second, we verify the Statechart diagram for each legacy program against its local properties. Third, for each adaptation from P_i to P_j identified in the requirements analysis, we design an adaption model, i.e., adaptive states and transitions from the source model M_i to the target mode M_j. Finally, we translate the adaptation models into Promela models to verify global invariants.

3.2.1 Generate state diagrams

We use a metamodel-based technique [5] to generate the state-diagram model M_i for each legacy program P_i. This technique had been previously proposed for formalizing a subset of UML diagrams in terms of different formal languages, including Promela. It is also generally applicable to formalizing the transformation of programs from one language to another. In order to apply this technique, we first define the metamodels for UML diagrams and for the legacy code, Java in this case. Then we define the rules for the translation from Java programs to UML models in terms of the metamodels. After the rules are defined, the translation from Java programs to UML models can then be performed mechanically by a developer and can potentially be automated. We have developed rules for translating the subset of Java that is relevant to our mobile computing and other applications of study. Developing the rules for translating the full Java language is non-trivial; ongoing investigations are underway by other groups [17].

3.2.2 Verify local properties for assurance

Next, we analyze the UML design models against local properties. We use the Hydra tool suite to transform the UML models to Promela models. Then we use the Spin model checker [18] to verify the Promela models against the local properties specified in requirements analysis. Violations of local properties by the models may indicate one or more of the following cases: (1) The legacy programs are initially incorrect, (2) the UML models are not generated faithfully according to legacy programs, or (3) the local properties are specified incorrectly. The corresponding erroneous artifacts must be revised until the models conform to the properties.

3.2.3 Design adaptation models

After the UML models are generated, we design an adaptation model for each required adaptation. Assume the program is required to adapt from running P_i (the source) to running P_j (the target), and the corresponding Statechart diagrams are M_i and M_j, respectively. We create an adaptation Statechart model from M_i to M_j by adding adaptive states and transitions such that the global invariants are preserved before, during, and after adaptation. We apply the MASD process to Statechart diagrams by performing the following tasks: (1) Identify the *quiescent states* in the source model, i.e., the states from which the adaptation may safely start; (2) identify the *entry states* in the target model, i.e., the states in which the adaptation completes; (3) determine the state transformation from the quiescent states to the entry states.

Previously [9], we argued that the adaptation model design is considered correct if and only if the model satisfies the global invariants specified in the requirements analysis. Although theoretically, the quiescent states and the entry states can potentially be any states in the models, certain

heuristics can be followed to keep the design clean and simple. First, since an adaptation can only start from a quiescent state, quiescent states must be on paths that the program frequently executes,[1] otherwise, there may be a long delay before the adaptation may start. Second, the conditions for the quiescent states must be kept simple. For example the conditions at the entry point of a loop are usually simpler than the conditions in the body of the loop. Example conditions include loop invariants, pre/post conditions, etc. Similarly, conditions for the entry states in the target program need to be simple as well. However, the entry states are not required to be on a frequently executed path.

The adaptive states and transitions usually include saving the states of the source model, transforming states of the source model to states of the target model, and restoring the states in the target model. The quiescent/entry states information that needs to be saved/restored are usually the values of those *live variables*, i.e., those that have been defined and may be used before they are redefined in the model. A state transformation defines a function from the set of variables in the quiescent state to the set of variables in the entry states. A necessary condition for a valid state transformation function is that the output must satisfy the conditions for the entry states given that the input satisfies the conditions for the quiescent states.

3.2.4 Verify global invariants for assurance

An adaptation model must be verified against the global invariants. The process is similar to that used for verifying the local properties. Violations of the global invariants indicate that the adaptation models or the global invariants are incorrect. In either case, we must return to previous steps to revise the corresponding artifacts until the global invariants are satisfied by the model.

3.3 Implementation of Adaptive Software

After the adaptation model is constructed, we implement the model in Java to enable the legacy code to be adaptive. We assume that each non-adaptive legacy program P_i is initially encapsulated in a Java class. First we create an adaptive Java class such that the class implements the adaptive behavior described by the adaptation model. Second, we replace invocations to the constructors of the non-adaptive classes in the legacy code with those of the adaptive class using an aspect-oriented technique. The remainder of the legacy program remains unchanged.

[1]The frequency may be monitored by instrumentation of the source code.

3.3.1 Construct adaptive classes

We implement adaptive programs by systematically realizing the adaptation models. Since the UML models are initially generated from the programs, we assume the traceability links between the UML models and the Java programs are already created by the generation process. We identify the locations and conditions in the source (resp. target) program that correspond to the quiescent (resp. entry) states in the adaptation model. At the locations corresponding to the quiescent state, we insert code to test whether an adaptation request has been received. If so, then the source program execution will be suspended and a state object comprising the current state information will be created. The state object is then transferred to the location in the target program using a cascade adaptation mechanism [15]. During the transfer, the state object is transformed from a state object for the source program to a state object for the target program. At the location in the target program, the state of the target program is restored from the state object and the execution is resumed from the point in the target program. We have developed a *cascade adaptation technique* to handle the state transformation from the source to the target in Java, which is omitted due to space constraints [15]

3.3.2 Enable adaptation in legacy code

To enable adaptation in legacy programs, we replace calls to the constructors of the non-adaptive classes with those of the adaptive class. Manually identifying the construction statements in the legacy code and modifying the code directly is undesirable. First, there may be numerous locations where the objects are constructed, making the manual approach tedious and error prone. Second, the adaptation concern will be entangled with the legacy code, making future maintenance difficult. Therefore, we apply the aspect-oriented technique to perform the code replacement. We define *pointcuts* to identify the calls to the constructors of the non-adaptive class. Then we use an *around advice* to replace them with calls to constructors of the adaptive class. The objects of the adaptive class then can be used throughout the legacy program in the same way as for the non-adaptive objects, except that they are capable of performing the designed adaptation. By using the aspect-oriented approach, we do not directly modify the legacy code, thus separating the adaptation concerns from the non-adaptation concerns.

4 Conclusions and Future Work

In this paper, we introduced an approach to transform non-adaptive legacy software into adaptive software with

assurance. Our approach leverages UML diagrams and formal techniques to provide assurance in adaptation, i.e., satisfying local properties and global invariants. In order to enable assured adaptation, our approach introduces software engineering activities on three layers: the implementation layer, the UML layer, and the formal language layer. The design of the adaptive programs is performed on the UML layer. The analysis for correctness is performed on the formal languages layer. The final adaptive programs are generated for the implementation layer. The transformation of different artifacts between these layers is accomplished by using the metamodel-based technique. In the implementation, adaptation concerns and non-adaptation concerns are disentangled by using an aspect-oriented programming approach.

We have developed guidelines for the transformation between different types of artifacts using the metamodel-based technique. So far, the guidelines are being systematically followed by the developers, however, they are amenable to automation. Our future work will focus on the automation of the translations between artifacts in the proposed technique. We have also initiated studies of using more design patterns to achieve more flexible architectures in the adaptive software implementation.

References

[1] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *IEEE Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[2] S. M. Sadjadi and P. K. McKinley, "Using transparent shaping and web services to support self-management of composite systems," in *Proc of the 2nd IEEE Inter. Conf. on Autonomic Computing*, 2005.

[3] J. P. Sousa and D. Garlan, "Aura: an architectural framework for user mobility in ubiquitous computing environments," in *Proc. 3rd Working IEEE/IFIP Conference on Software Architecture*, 2000.

[4] J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger, "A survey of self management in dynamic software architecture specifications," in *Proc. Inter. Workshop on Self-Managed Systems*, pp. 28–33, 2004.

[5] W. E. McUmber and B. H. C. Cheng, "A general framework for formalizing uml with formal languages," in *ICSE'01*, pp. 433–442, 2001.

[6] Z. Yang, B. H. C. Cheng, K. Stirewalt, M. Sadjadis, J. Sowell, and P. McKinley, "An aspect-oriented approach to dynamic adaptation," in *Proc of the ACM SIGSOFT Workshop on Self-Healing Systems*, 2002.

[7] B. Redmond and V. Cahill, "Supporting unanticipated dynamic adaptation of application behaviour," in *Proc. of the 16th European Conf. on Object-Oriented Programming*, 2002.

[8] J. Zhang, J. Lee, and P. K. McKinley, "Optimizing the Java pipe I/O stream library for performance," in *Proc. 15th Inter. Workshop on Languages and Compilers for Parallel Computing*, 2002.

[9] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *ICSE'06*, 2006.

[10] R. Allen, R. Douence, and D. Garlan, "Specifying and analyzing dynamic software architectures," in *Proc. 1998 Conf. on Fundamental Approaches to Software Engineering*, 1998.

[11] J. Kramer and J. Magee, "Analysing dynamic change in software architectures: a case study," in *Proc. 4th IEEE Inter. Conf. on Configuratble Distributed Systems*, 1998.

[12] C. Canal, E. Pimentel, and J. M. Troya, "Specification and refinement of dynamic software architectures," in *Proc. of the TC2 First Working IFIP Conference on Software Architecture*, pp. 107–126, 1999.

[13] A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu, "Towards requirements-driven autonomic systems design," in *Proc of 2005 Workshop on Design and Evolution of Autonomic Application Software*, 2005.

[14] P. Bertrand, R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde, "GRAIL/KAOS: an environment for goal drivent requirements engineering," in *ICSE'98*, IEEE Computer Society, 1998.

[15] J. Zhang and B. H. Cheng, "Re-engineering legacy systems for assured dynamic adaptation," Tech. Rep. MSU-CSE-07-11, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, February 2007.

[16] A. van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Proc. 5th IEEE Inter. Symposium on Requirements Engineering*, p. 249, 2001.

[17] R. Kollmann, P. Selonen, E. Stroulia, T. S. ä, and A. Z. ndorf, "A study on the current state of the art in tool-supported uml-based static reverse engineering," in *Proc. 9th Working Conf. on Reverse Engineering*, pp. 22–32, 2002.

[18] B. N. Bershad *et al.*, "Spin - an extensible microkernel for application-specific operating system services," tech. rep., Dept. of Computer Science and Engineering, University of Washington, 1994.