Model-Based Development of Dynamically Adaptive Software *

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 should dynamically adapt its behavior at run-time in response to changing conditions in the supporting computing and communication infrastructure, and in the surrounding physical environment. In order for an adaptive program to be trusted, it is important to have mechanisms to ensure that the program functions correctly *during* and *after* adaptations. Adaptive programs are generally more difficult to specify, verify, and validate due to their high complexity. Particularly, when involving multi-threaded adaptations, the program behavior is the result of the collaborative behavior of multiple threads and software components. This paper introduces an approach to create formal models for the behavior of adaptive programs. Our approach separates the adaptation behavior and non-adaptive behavior specifications of adaptive programs, making the models easier to specify and more amenable to automated analysis and visual inspection. We introduce a process to construct adaptation models, automatically generate adaptive programs from the models, and verify and validate the models. We illustrate our approach through the development of an adaptive GSM-oriented audio streaming protocol for a mobile computing application.

Categories and Subject Descriptors: D.2.4 [SOFTWARE ENGINEERING]: Software/Program Verification - Formal methods, Model checking, Reliability, Validation

General Terms: Design, Verification, Reliability

Keywords: Dynamic Adaptation, Reliability, Autonomic Computing, Global Invariants, Formal Specification, Verification

1. INTRODUCTION

Increasingly, software needs to dynamically adapt its behavior at run-time in response to changing conditions in the supporting computing, communication infrastructure, and in the surrounding physical environment [1]. In order for an adaptive program to be trusted, it is important to have mechanisms to ensure that the program functions correctly *during* and *after* adaptations. Adaptive programs are generally more difficult to specify, verify, and validate due to their high complexity. Particulary, when involving multi-threaded adaptations, the program behavior is the result of the collaborative behavior of multiple threads and software components. Adaptations require the adaptive actions of all the components and threads to be taken in a coordinated fashion. Formal models can be used to mitigate the complexity. This paper introduces an approach to create formal models for the behavior of adaptive programs. Our approach separates the adaptation behavior and non-adaptive behavior specifications of adaptive programs, making the models easier to specify and amenable to automated analysis and visual inspection. We introduce a process to construct adaptation models, automatically generate adaptive programs from the models, and verify and validate the models.

A recent survey [2] described numerous research efforts that have proposed ways to formally specify dynamically adaptive programs in the past several years. Graph-based approaches model the dynamic architectures of adaptive programs as graph transformations [3, 4, 5]. Architecture Description Language (ADL)-based approaches model adaptive programs with connection and reconnection of connectors [6, 7, 8]. Generally, these approaches have focused on the structural changes of adaptive programs. Few efforts have formally specified the behavioral changes of adaptive programs. A few exceptions include those that use process algebras to specify the behavior of adaptive programs [9, 10, 11]. However, they share the following drawbacks: (1) Portions of the adaptation-specific specifications are entangled with the non-adaptive behavior specifications; (2) They do not support the specification of state transfer, so that the target behavior must start from the initial state, thus making adaptations less flexible; (3) These techniques are specific to the type of formalism (i.e., process algebra), making them potentially difficult to be extended to other types of formalisms and the corresponding analysis.

In this paper, we propose a model-driven process to the development of dynamically adaptive programs. Compared to existing work, our approach has the following novel features: (1) Our approach separates the model specifying the non-adaptive behavior from the one specifying the adaptive behavior, thus making the model more amenable to human inspection and automated analysis; (2) We use global invariants to specify properties that should be satisfied by adaptive programs regardless of adaptations. These prop-

*This work has been supported in part by NSF grants EIA-0000433, EIA-0130724, ITR-0313142, and CCR-9901017, 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'06, May 20–28, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005 ...$5.00.

erties are ensured throughout the program's execution by model checking; (3) Our specification approach supports state transfer from the *source program* (i.e., the program before adaptation) to the *target program* (i.e., the program after adaptation), thereby potentially providing more choices of states in which adaptations may be safely performed; (4) Our approach is generally applicable to many different state-based modeling languages. We have successfully applied our approach to several state-based modeling languages, including process algebras (e.g., Petri nets), UML state diagrams, and model-based languages (e.g., Z); (5) We also introduce a technique that uses the models as the basis for automatically generating executable prototypes, and ensures the model's consistency with both the high-level requirements and the adaptive program implementations.

This paper focuses on the *behavior* of adaptive programs. We explicitly identify and specify the key properties of adaptation behavior that are common to most adaptive programs, regardless of the application domain, the programming language, or the adaptation mechanism. We define the *quiescent states* (i.e., states in which adaptations may be safely performed) of an adaptive program in the specific adaptation context, including the program behavior before, during, and after adaptation, the requirements for the adaptive program, and the adaptation mechanism. This definition leads to precise and flexible adaptation specifications.

We have successfully applied our approach to adaptive mobile computing applications, including the development of an adaptive GSM-oriented audio streaming protocol [12]. This protocol had been previously developed without our technique, where the developers had found the program logic to be complex and error-prone. Our approach significantly reduced the developer's burden by separating different concerns and utilizing automated model construction and analysis tools, and thus improved both the development process time and reliability. The remainder of the paper is organized as follows: Section 2 gives background on Petri nets and the GSM-oriented audio streaming protocol. Section 3 introduces the formal representation of adaptive programs. Section 4 describes our approach for constructing and analyzing models for adaptive programs. Section 5 outlines the generation of adaptive programs based on the models. Section 7 discusses related work, and Section 8 concludes the paper.

2. BACKGROUND

In this paper, we illustrate our adaptation specification process by modeling a GSM-Oriented audio streaming protocol with Petri nets. This section briefly overviews Petri nets and the GSM-oriented audio streaming protocol.

2.1 Petri Nets

Petri nets are a graphical formal modeling language [13, 14], where a Petri net consists of *places*, *transitions*, and *arcs*. Places may contain zero or more tokens. A state (i.e., a *marking*) of a Petri net is given by the number of tokens in each place. Places are connected to transitions by *input arcs* and *output arcs*. *Input arcs* start from places and end at transitions; *output arcs* start from transitions and end at places. The places incident on the input (or output) arcs of a transition are the *input places* (or *output places*) of the transition. Transitions contain *inscriptions* describing the *guard conditions* and associated *actions*. A transition is *enabled* if the tokens in its input places satisfy the guard condition of the transition. A transition can be *fired* if it is enabled, and firing a transition involves consuming the tokens from its input places, performing the associated actions, and producing new tokens in its output places. An *execution* of a Petri net comprises a sequence of transitions. Interactive executions of Petri nets are called *token games*.

Since their introduction, Petri nets have been extended in many ways. *Coloured Petri nets* allow the tokens to be distinguished by their colors (or types), making it more convenient to model data in software. The inscriptions on arcs of a coloured Petri net specify the numbers and types of tokens to be taken from (or put into) the incident places when the transitions are fired.

Numerous tool suites have been developed to support graphical net construction, simulations, token games, and analyses. Among these tools, *MARIA* [15] is a coloured Petri nets analysis tool that supports model checking for Linear Temporal Logic (LTL) properties. *Renew* [16] is a coloured Petri nets tool suite that supports graphical net construction, automated simulation, and token games. Furthermore, Renew supports a synchronous communication mechanism that allows Petri net models to communicate with each other and with other Java applications.

2.2 GSM-Oriented Encoding/Decoding

GSM-oriented audio stream encoding and decoding protocol is a signal processing-based forward error correction protocol [17, 12]. The protocol is used to lower delay and overhead in audio data transfer through lossy network connections, especially in a wireless network environment. In this protocol, a lossy, compressed data segment of packet i is piggybacked onto one or more subsequent packets, so that even if packet i is lost, the receiver still can play the data of i at a lower quality as long as one of the subsequent encodings of i is received. This protocol takes two parameters, c and θ. The parameter c describes the number of consecutive packets onto which an encoding of each packet is piggybacked. The parameter θ describes the offset between the original packet and its first compressed version. Figure 1 shows a GSM-oriented audio streaming protocol example with $\theta = 1$, $c = 2$. In order to accommodate packet loss changes in the network connection, we dynamically switch among components that implement different θ and c values.

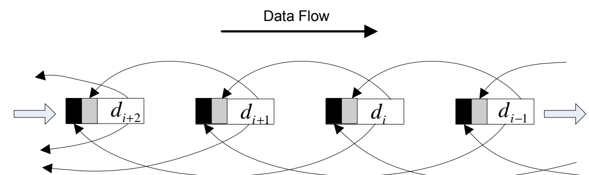

Figure 1: GSM-oriented encoding and decoding.

3. STATE-MACHINE REPRESENTATION FOR ADAPTIVE PROGRAMS

In this section, we introduce a formal representation of adaptive programs. In general, *a program* can be represented by a state machine [18] (finite state automaton) that exhibits certain behavior and operates in a certain domain (its input space) [19]. A dynamically adaptive program operates in different domains, changes its behavior at runtime in response to changes of the domain. An adaptive program is a program whose state space can be separated into a number of disjoint regions (*programs*), each of which exhibits a different *steady-state* behavior [9], and operates in a different domain. The states and transitions connecting one region to another are *adaptation sets*.

We term the properties that should be satisfied by the adaptive program in each individual domain as the *local properties* for the domain. The properties that should be satisfied by the adaptive program throughout its execution,

regardless of the adaptations, are called *global invariants*. Since global invariants should also be satisfied by adaptive programs in each individual domain, they must be implied by local properties.

An adaptive program usually contains multiple programs and multiple adaptation sets connecting these programs, making it potentially complex to analyze. To reduce the complexity, we isolate the adaptation problem by focusing on the adaptation behavior starting from one program, undergoing one occurrence of adaptation, and reaching a second program. This type of adaptation behavior is represented by *simple adaptive programs*. A simple adaptive program contains a source program, a target program, and an adaptation set connecting the source program to the target program. Figure 2 shows a simple adaptive program where S is the source program, T is the target program, and M is the adaptation set from S to T. In this paper, we specify the source program and the target program with a *source model* and a *target model*, respectively, and then specify an *adaptation model*, i.e., the model for the adaptation set connecting the source model to the target model. A general adaptive program can be considered as the union of one or more simple adaptive programs [20].

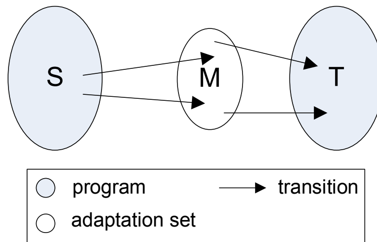

Figure 2: A simple adaptive program

We propose a general specification process for state-based modeling languages. We also introduce the analyses that may be performed to ensure the consistency among the models, high-level requirements, and low-level implementations. We illustrate the process with Petri nets and a working adaptive GSM-oriented protocol example. Although we illustrate our approach with Petri nets, our approach extends to other modeling languages.

Assume we are given a problem in terms of a set of high-level goals [21, 22] and a set of execution domains D. We are required to develop an adaptive program to solve the problem, i.e., to achieve the high-level goals, when the inputs of the program change their domains in D at run time. We propose the following adaptive program development process:

(1) Use a high-level specification languages, e.g., temporal logic, to specify global invariants. These global invariants usually contain safety and liveness constraints.

(2) Enumerate the different domains in D. Describe the conditions in which the program is required to execute in each domain (e.g., data loss conditions of a communication channel).

(3) Use high-level specification languages, e.g., temporal logic, to specify the local properties (the *local requirements*) of the program in each domain. The specifications should satisfy the high-level goals under the conditions of the domain.

(4) Build a state-based model for the program in each domain. Simulations and verifications may be applied to verify and validate the models against the local properties of the domain.

(5) Enumerate possible dynamic changes of domains and build adaptation models for the adaptations of the program from one domain to another. Simulations and verifications may be applied to validate the models against the global invariants. Moreover, the adaptations should lead to the target models eventually.

(6) These state-based models can be further used to generate rapid prototypes or serve to guide the development of adaptive programs. They can also be used to generate test cases and verify execution traces.

In this paper, we assume that steps (1) through (3) have been done properly using existing techniques [19, 20], and focus on steps (4) through (6). When errors are found in a given step, developers may be required to go back to an earlier step to correct the errors.

4. OUR SPECIFICATION APPROACH

In this section, we introduce the general process to model an adaptive program in a state-based modeling language, accompanied by a concrete GSM-oriented audio streaming protocol modeled with Petri Nets.

4.1 Motivating Adaptation Scenario

The configuration of the system is shown in Figure 3. A desktop computer, running a sender program, records and sends audio streams to hand-held devices (e.g., iPAQ), running receiver programs, through a lossy wireless network. We expect the system to operate in two different domains: the domain with a low loss rate and the domain with a higher loss rate. The loss rate of the wireless connection changes over time and the program should adapt its behavior accordingly: When the loss rate is low, the sender/receiver should use a low loss-tolerance and low bandwidth-consuming encoder/decoder; and when the loss rate is high, they should use a high loss-tolerance and consequently high bandwidth-consuming protocol. Specifically, in this paper, we describe the simple adaptive program that initially uses GSM(1,2) encoding/decoding when the loss rate is low (where $\theta = 1$, $c = 2$); when the loss rate becomes high, it dynamically adapts to using GSM(1,3) encoding/decoding.

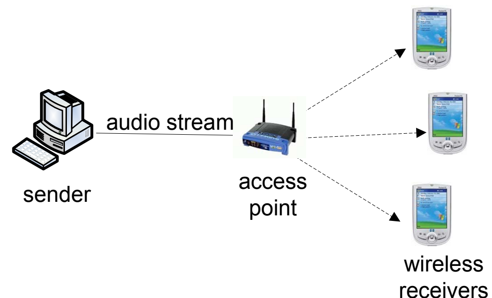

Figure 3: Audio streaming system connection

4.2 Construct Models for Source and Target

This section corresponds to Step (4). Assume that we already have the local requirements for the source domain (the *source requirements*) and for the target domain (the *target requirements*). We need to build a model for the

source domain (the *source model*) and a model for the target domain (the *target model*). The source and target models should not include information about each other, or about the adaptation. The source and target models should be verified against the local requirements for the source and target domains, respectively.

This step is illustrated by example as follows. Assume we have identified the source domain S (the domain with low loss rate) and the target domain T (the domain with high loss rate). In this example, we use two LTL temporal operators: $\Box$ and $\Diamond$. Intuitively, the formula "$\Box P$", where P is a property, means that the property "P" is always true during an execution; the formula "$\Diamond P$" means that "P" is eventually true in an execution.
The requirements R_S for the source domain are:

- **Sender liveness:** The sender should read packets until the data source is empty (i.e., the data source is eventually empty), and the sender should always eventually send a packet if it reads a packet. In LTL formula:

  $\Diamond(dataSource = empty) \wedge \Box(read(x) \rightarrow \Diamond send(x))$ [1]

- **Receiver liveness:** The receiver should always decode data once a new packet is received. In LTL formula:

  $\Box(receive(x) \rightarrow \Diamond decode(x))$

- **Loss tolerance:** The sender/receiver should use a protocol that tolerates 2-packet loss. In LTL formula:

  $(\Box lossCount <= 2) \rightarrow (\Box \neg lose(x))$

The requirements R_T for the target domain has the same set of properties except the loss tolerance constraint as follows:

- **Loss tolerance:** The sender/receiver should use a protocol that tolerates 3-packet loss. In LTL formula:

  $(\Box lossCount <= 3) \rightarrow \Box(\neg lose(x))$

To model the program for S, we build a Petri net for a GSM(1,2) encoder on the sender side and a Petri net for GSM(1,2) decoder on the receiver side. Figure 4 shows the sender net (elided). The circles represent places and boxes represent transitions. The white circles represent places that are not part of the sender model, i.e., shared by either the source and the target sender nets, or by the sender and the receiver nets, or by both. In the net, the place dataSource contains a sequence of data tokens. The readData transition removes a token from the dataSource place, and puts the original data in the inputData place and a GSM compressed data in the dataX place. The shiftX and shiftY transitions shift the encoded data in a buffer represented by the places dataX, dataY, and dataZ. The encode transition takes the current input data and two previously compressed data from the places dataY and dataZ, and outputs a GSM(1,2) data packet in the encodedData place. The send transition sends the encoded packet to the network place.

Figure 5 shows the receiver net (elided). The receiveData transition receives the audio packets from the network and puts the input packet in the inputData place. The decodeInput transition takes the input packet from the inputData place, extracts data from the original data segment, and puts the data in the bufferedData place. The decodeBuffer transition extracts the compressed data segments from the input packet, and updates the data in the bufferedData place. The outputData transition takes decoded data from the bufferedData, and then outputs the data to the sink. In a separate model (not included in this paper due to space constraints), we build the Petri net for the network, which is the glue specification that connects the sender and the receive nets. We specify two types of networks: lossless and lossy.

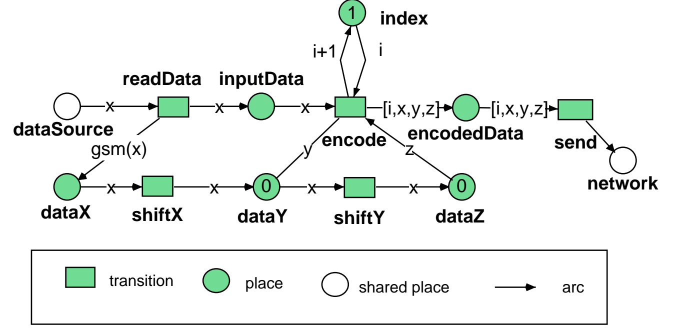

Figure 4: Sender source net

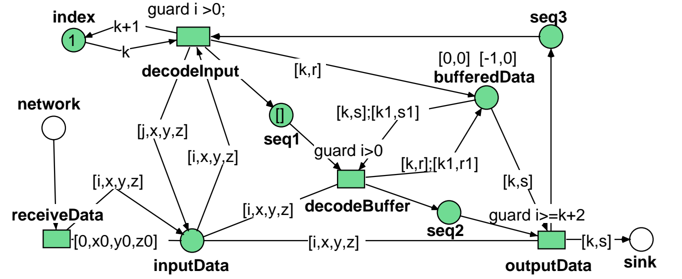

Figure 5: Receiver source net

After building the source models, we first play token games with the models to visually validate the models. If we find errors with the models, then we revise the models until the models pass the visual validation. Then we run model checking to verify these models against the high-level local requirements of the source domain S, including the safety, liveness, and loss-tolerance constraints. We revise the models until they pass the model checking analysis.

To model the program for the target domain T, we build separate Petri nets for a GSM (1,3) encoder on the sender side and a GSM (1,3) decoder on the receiver side. Figures 6 and 7 show the sender net and the receiver net, respectively.[2] The modeling, validation, and verification process is similar to that used for the source model.

4.3 Construct adaptation models

This section describes Step (5), the process to model the adaptation behavior from the source domain to the target domain. Three types of adaptive behavior are usually seen in adaptive programs [20]: *one-point adaptation*, *guided adaptation*, and *overlap adaptation*. We introduce the modeling technique for each of these types of adaptation. For each type of adaptation, we first introduce a general specification

[1]Strictly speaking, the notations of $read(x)$ and send(x) are predicates. However, LTL requires the underlying logic to be propositional. Here we implicitly employ the *2-order data abstraction* introduced by Dwyer and Pasareanu [23], which converts predicates to propositions by using constant values to represent arbitrary values.

[2]By comparing the source model to the target model, one can notice that the source model can be systematically extended to construct target models for GSM-oriented protocol of various parameters.

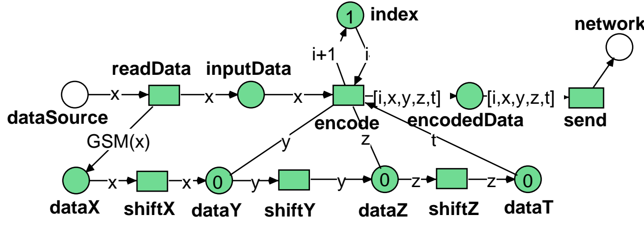

Figure 6: Sender target net

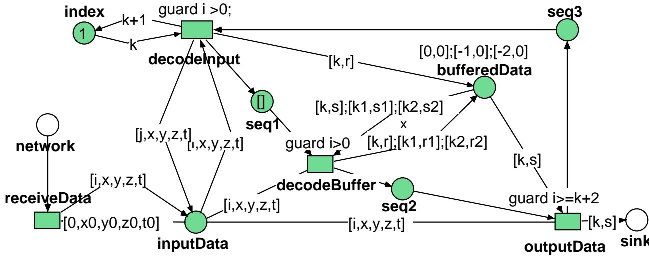

Figure 7: Receiver target net

approach for state-based modeling languages, then we instantiate the approach with Petri nets and apply it to the GSM-oriented protocol example.

4.3.1 One-point adaptation

With *one-point adaptation*, at one state during the source program's execution, the source behavior should complete, and the target behavior should start [20]. The adaptation process completes after a single transition. The major tasks for one-point adaptation are to identify the states that are suitable for adaptation and define adaptive transitions from those states.

The term *quiescent states* are commonly used to refer to those states suitable for adaptations in the literature. They are usually identified as the "stateless" states of a program, e.g., states equivalent to initial states. However, in some programs, reaching such states may not occur in a reasonably short period of time, causing the program execution to block. Thus, this type of definition for quiescent states is not suitable for changes that require prompt adaptation responses, such as fault tolerance, error recovery, attack detection, etc. Furthermore, such a quiescent state definition is not sufficient to ensure the correctness of adaptation in the absence of the requirements to be achieved by the adaptation.

In this paper, we argue that the quiescent states of an adaptive program should be defined in the context of the adaptation constrained by the program behavior before, during, and after adaptation, and the global invariants that the adaptive program should satisfy throughout its execution. A state of the source program, s, is a quiescent state, if and only if we can define a *state transformation function*, f, such that there exists a state, t, in the target program, $t = f(s)$, and any execution paths that include the $s \rightarrow t$ adaptive transition do not violate any global invariants.

The set of quiescent states is determined by the state transformation from s to t that satisfies the global invariants. Generally speaking, the more quiescent states we can identify, the more flexible the adaptation is, i.e., the more states from which we may perform adaptation. Potentially, all states of the source model can be quiescent states, but that would require us to define a complex state transformation function. Therefore, we should balance the complexity of the transformation function and the flexibility of the adaptation.

We use Petri nets to illustrate this idea. We define an "adapt" transition to model the set of adaptive transitions. The "adapt" transition connects the source net to the target net: All the input places of the transition are from the source net, and all the output places are from the target net. When the "adapt" transition is fired, it performs the state transformation by consuming the tokens in the source model and generating tokens in the target model. The quiescent states of the Petri net are those markings that enable the "adapt" transition, which can be restrained by the guard conditions of the transition. More than one "adapt" transition can be defined similarly, each identifying a different set of quiescent states and defining a different state transformation function upon this set. The source net and the target net, connected by the "adapt" transition, is the adaptation model. We use token games and model checking to validate and verify the adaptation model against the global invariants properties: If violations are found, then we should revise the models and/or the properties.

In the GSM-oriented audio streaming protocol example, the sender or the receiver adaptation alone can be considered one-point adaptation. The global invariants for the adaptive sender and receiver are specified as follows:

- **Sender global invariant:** The sender should read packets until the data source is empty, and the sender should always eventually send a packet if it reads a packet. In LTL,
  $\Diamond(dataSource = empty) \wedge \Box(read(x) \rightarrow \Diamond send(x))$
- **Receiver global invariant:** The receiver should always decode data once a new packet is received. In LTL,
  $\Box(receive(x) \rightarrow \Diamond decode(x))$

The adaptation model for the sender is shown in Figure 8. The enabling condition of the "adapt" transition of the sender identifies the quiescent states to be "after encoding a packet and before sending the packet, and after the data in the compressed data buffer have been shifted to the next location". The "adapt" transition directly moves the tokens from dataY and dataZ to the corresponding places in the target. The token in dataT in the target model is generated from the encoded packet in encodedData of the source by taking the last piggybacked data segment z.

Figure 9 shows the adaptation model for the receiver. The quiescent states of the receiver are identified by the adapt transition of the receiver. That is, "after firing the decodeInput transition and before firing the decodeBuffer transition, and when the upcoming input packet is encoded by the target encoder". Upon receiving the packet, the model adapts to the target receiver net, and the state transformation is defined by the output places and inscriptions on the arcs.

We ran token games on the sender and receiver adaptation models, and we validated that they reflect our purpose for the software. Last, we model checked these models against the global invariants. Finally, we concluded that these properties are all satisfied. Therefore, we believe the models have been constructed correctly.

4.3.2 Guided adaptation

Guided adaptation means that when the source program receives an adaptation request, it enters a restricted mode, in which some functionalities of the program are usually blocked [20]. Entering the restricted mode ensures that the program will reach a quiescent state, from which a one-point adaptation takes the program to a target program state space.

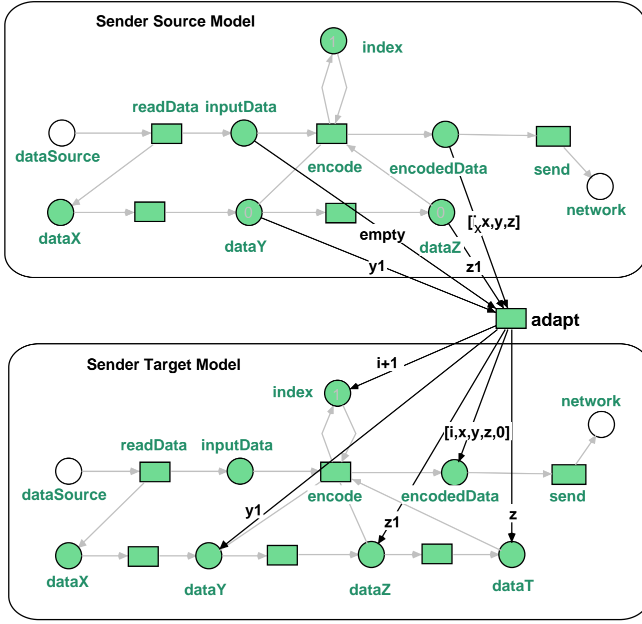

Figure 8: Sender adaptation net

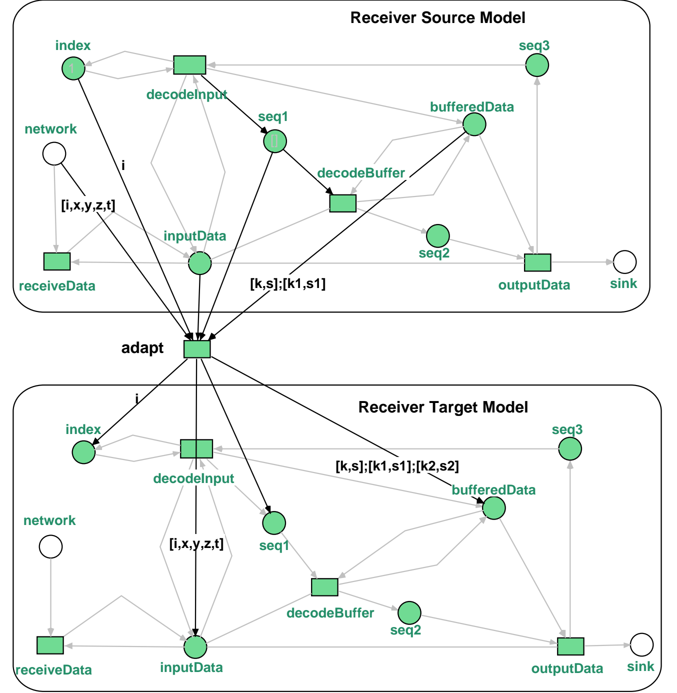

Figure 9: Receiver adaptation net

To specify a guided adaptation, we should determine the functionalities that should be blocked in the restricted mode, and identify the quiescent states of the program in the restricted mode. Blocking functionalities is achieved by removing transitions from the source program. Let the source model be M_S, the target model be M_T, and the restricted source model be M'_S. M'_S must share the same set of states with M_S, but M'_S has only a subset of the transitions of M_S. M_S and M_T can be constructed in the same way as in one-point adaptation. M'_S can be constructed by first copying M_S and then removing transitions or strengthening the firing conditions of transitions that may otherwise prevent the program from reaching a quiescent state.

We next define the state transformations from states in M_S to states in M'_S and from quiescent states in M'_S to states in M_T. As M'_S shares all the states with M_S, the state transformation from M_S to M'_S is trivially an identity function (a function that maps an element to itself) on the domain of all the states in M_S. The approaches to define quiescent states of M'_S and the state transformation from M'_S to M_T are the same as those in one-point adaptation.

The way M'_S is constructed ensures that any execution path of M'_S is also an allowable execution path of M_S, which implies that as long as M'_S does not cause deadlock, M'_S satisfies all safety and liveness constraints that M_S does. The properties we need to verify about M'_S are that it does not reach a deadlock state before it eventually reaches a quiescent state, and that the adaptation model constructed by M_S, M'_S, and M_T should satisfy the global invariants.

The guided adaptation can be illustrated with the GSM-oriented adaptive sender model. Assume the quiescent states of the sender are identified by the adapt transition in Figure 8, which requires the inputData place to be empty and the encodedData place to be non-empty, and the encoded data in dataY and dataZ have already shifted one location. The semantics of Petri nets determines that the order for firing the send transition and shifting the data in dataY and dataZ is non-deterministic. It might be the case that the send transition is always fired before shifting the data, rendering the quiescent states unreachable. To deal with this problem, we construct a net N by first copying the source net, then removing the send transition from the net. Figure 10 illustrates the adaptation from the sender source net to the sender restricted source net. The **restrict** transition represents a total identity function from the source to the restricted source N. The adapt transition from N to the target net is similar to that in Figure 8. Given the verification we have already performed, the only additional property we need to verify is that N will eventually reach a quiescent state. (Note, the way N is constructed guarantees that all other properties verified before are still valid.) We model checked the model in Figure 10 and concluded that the property holds.

4.3.3 *Overlap adaptation*

Overlap adaptation means the source and the target behavior overlap [20]. At point A, the target behavior starts, and at a later point B, the source behavior stops. During the course between A and B, the system exhibits both the source and the target behavior. For overlap adaptation, the source to target adaptations are accomplished by a sequence of adaptation transitions that are performed one after another. The target program starts to execute after the first adaptive transition, and the source program completes after the last adaptive transition. Overlap adaptation is typical in multi-threaded programs, where different threads adapt to the target. Each thread performs a one-point adaptation or guided adaptation at different times, and the combined result becomes overlap adaptation.

Overlap adaptation is more complex than one-point adaptation in the sense that we not only need to determine the state transformation functions of each one-point or guided adaptation it comprises, but also the coordination among these adaptations. Example coordination relationships among these adaptations include precedence relationship, cause-effect relationship, parallel relationship, etc. The key task in modeling overlap adaptations is to define how these multiple adaptations should coordinate with each other in order to satisfy global invariants. In addition to satisfying explicitly

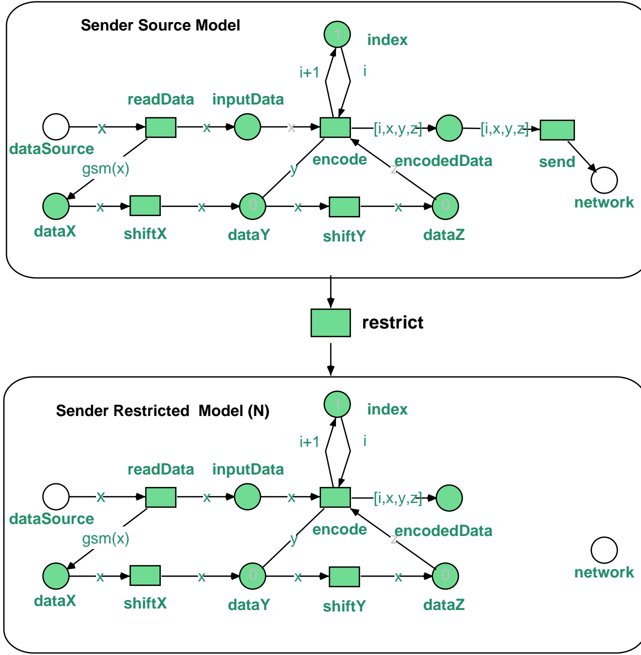

Figure 10: Sender restricted source net

specified global invariants, an adaptive program should also satisfy an *adaptation integrity constraint*: **Once the adaptation starts, it should complete**, i.e., the adaptation should finally reach a state of the target program. Violations of this constraint result in an inconsistent state of the program that is not designed for the target domain, and we have no means to ensure its correctness.

The example shown in Figures 8, 9, and 10 is, in fact, an entire model with overlap adaptation. After the sender has adapted to the target domain, the receiver still remains in its source domain. The adaptation starts when the sender `adapt` transition is fired, and ends when the receiver `adapt` transition is fired. The adaptation of the sender and the receiver has a cause-effect relationship: The receiver's adaptive transition is triggered by a packet sent by the adapted sender. By composing the sender and the receiver adaptation as an overlap adaptation, we are able to specify the following two additional constraints:

- **GSM example loss-tolerance global invariant:** The adaptive program should tolerate 2-packet loss throughout its execution. In LTL,

  $(\Box lossCount <= 2) \rightarrow (\Box \neg lose(x))$

We used model checking to verify this property successfully.

- **GSM example adaptation integrity constraint:** If the sender adaptive transition is fired, then the receivers's adaptive transition will also eventually be fired. In LTL,

  $\Box(senderAdapted \rightarrow \Diamond receiverAdapted)$

We found errors when model checking the adaptation integrity constraint. By inspecting the counter example, we realized that in a rare case, if all the packets after the sender's adaptation are lost, then the receiver will not receive any packet encoded by the target sender, and thus the receiver will not adapt. We revised the model by using a reliable communication channel to send the first packet after sender adaptation, so that the receiver will be guaranteed to receive the packet. Note that it is generally possible to build a reliable communication channel on top of unreliable underlying infrastructure by using acknowledgement-based protocols. Using it to send audio-stream would incur a performance penalty. However, the penalty is negligible, if we use it to send only critical packets occasionally. We reran the model checking for the revised model against the adaptation integrity constraint and the result showed that the adaptation indeed runs to completion with the revised model.

4.4 Discussion

As described in Section 3, depending on the perspective and the level of abstraction in which the developers are interested, a source program, a target program, or an adaptation set may be adaptive itself. The above specification technique may be applied recursively to specify the internal structure of a program or an adaptation set. For the GSM-oriented protocol example, we may apply guided adaptation for the sender and one-point adaptation for the receiver, resulting in a more complex adaptation scenario.

For a general adaptive program with multiple programs and adaptation sets, we first divide the program into a number of simple adaptive programs, then specify each simple adaptive program individually. In our approach, we verify the global invariants for each simple adaptive program. We expect the global invariants to hold for all executions, including those with multiple occurrences of adaptations. We can prove that this is the case for all *point safety* and *point liveness* LTL formulae and their propositional compositions [24]. A *point safety* formula has the form $\Box\neg\eta$ where η is a point formula [25], i.e., a formula without temporal operators. A *point liveness* formula has the form $\Box(\alpha \rightarrow \Diamond\beta)$, where both α and β are point formulae. The global invariants discussed in this paper are all point liveness, point safety properties, or their propositional compositions.

5. REIFYING THE MODELS

An adaptive model is an abstraction of an adaptive program in the sense that a model is a projection of the program behavior on an interesting alphabet (i.e., transitions); it represents a partial view of a program in which we are interested. We explain this idea with the GSM-oriented audio streaming example. From the models we have built, we can identify four different programs (Figure 11): the source and the target programs P_S and P_T, and two intermediate programs P_1 and P_2. The model in Figure 8 describes the `adapt sender` adaptation projected onto the sender. The model in Figure 9 describes the `adapt receiver` adaptation projected onto the receiver. The model in Figure 10 describes the `restrict sender` adaptation projected onto the sender.

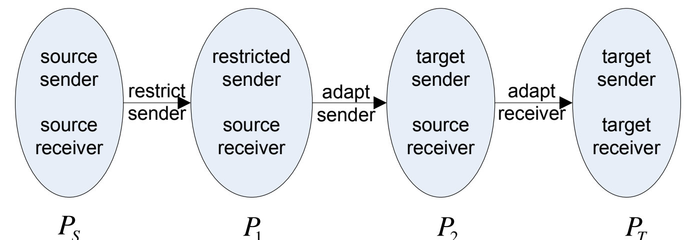

Figure 11: An adaptive program state machine

This section introduces Step (6), the approach to generate executable prototypes and develop code based on the models constructed in the previous section with the assistance of the Renew tool suite [16].

5.1 Rapid Prototyping

Renew [16] supports the specification of implementation-specific (Java) code in its transition inscriptions. When a transition is fired, the code associated with the transition will be executed. Model-driven approaches make the model in charge of the sequence in which the transitions are fired. By using this mechanism, we can generate rapid prototypes directly from the adaptive models, whose behavior has been verified. We map each transition to a Java function call, whose functionality is manually generated based on the input/output places, guard conditions, and other inscriptions of the transition. The `adapt` transition is mapped to an `adapt` function, which implements the necessary state transformation. The transition does not put any constraint on how the `adapt` function should be implemented. It could replace an old object representing the source program with a new object representing the target program, and transfer the state from the old object to the new object. Or alternatively, it could directly update the state of the object for the source program. In our implementation, we chose to use replacement of objects to achieve the adaptation.

Following the procedure introduced above, we have built the rapid prototype for the adaptive sender and adaptive receiver in Java, and executed the application. The transformation from the model to the prototype required a small amount of effort when compared to the time we spent to specify and verify the models. We have tested the execution results, and the rapid prototype executed as expected.

5.2 Model-Based Testing

Given the rapid prototype generated previously, the adaptive program can be easily designed and implemented. After the program is carefully implemented, we should ensure that the program is implemented correctly, i.e., it is a refinement of the models. The program should satisfy the following conformance constraints with the models:

1. Each transition in the model should have a corresponding *handler function* in the implementation.

2. The sequence in which the handler functions are called should be an allowable transition sequence (i.e., an execution) of the Petri nets model.

The first constraint can be easily verified syntactically. The second conformance constraint can be ensured by model-based testing techniques [26]. We use the synchronous communication channel mechanism for Java provided by Renew to invoke transitions in Petri nets: A transition is fired when it is invoked through the communication channel and it is enabled. We add an invocation of each transition of the Petri nets to the Java handler function of that transition. When the Java function is executed, the corresponding Petri net transition will be invoked as well. If the transition is also enabled at the time it is invoked , then the transition will be successfully fired, otherwise, deadlock will occur. When the Java program is executed, if the sequence in which these handler functions are invoked is an allowable sequence of the Petri nets, then the execution will complete successfully, otherwise, it will deadlock. With this approach we can test the conformance between the executions of the Java implementation and that of the Petri net models.

6. MODELING WITH OTHER LANGUAGES

Each different modeling language is more suitable for certain types of tasks than others. Based on the target tasks at hand, we may choose one language over another. For example, we chose Petri nets as the modeling language in this paper because it is most suitable to model the GSM-oriented audio streaming protocol, and that it has good tool support. Our proposed modeling approach is intended to be generally applicable to other state-based modeling languages. In this section, we briefly introduce the key steps to apply our technique to two other representative modeling languages: Z and UML state diagrams.

Z [27] is a model-based language that is well-suited for modeling abstract data. We can consider a Z model as a state machine where states are specified by schemas that describe abstract states, and transitions are specified by schemas that describe operations. Z can model adaptive programs that involve complex data operations, especially complex state transformation functions. To apply our modeling approach to Z, we first build a Z model for each execution domain of the program that satisfies the given high-level requirements. Then we define the adaptation from one domain to another with a set of "adapt" schemas, each of which defines a set of quiescent states and a state transformation function from these states. We have applied Z to the specification of the buffer operations of the sender and the receiver during adaptation in the GSM-oriented audio streaming protocol.

UML [28] is widely considered the *de facto* modeling languages in software industry due to its intuitive representation. We also applied our approach to UML state diagrams to specify the behavior of programs. We use a state diagram to specify the program behavior in each domain. Note that for a state diagram comprising concurrent partitions, a program state is represented by a set of states in the diagram. Therefore, we define the notion of quiescent *state cut* to represent a set of states that identify the program state in a state diagram suitable for adaptation. Then we define "adapt" transitions that connect a set of *state cuts* of one diagram to a set of state cuts of another to specify the adaptations among two different domains.

After the models are created, we can use existing tools to reason about the models for correctness properties. The analyses and model reification introduced in this paper may or may not apply to a specific modeling language of choice because they are supported and limited by available tool suites, which is beyond the scope of this paper.

7. RELATED WORK

The work presented in this paper has been significantly influenced by several related projects on formally specifying adaptive program behavior with process algebraic languages. For example, Kramer and Magee [10] have used Darwin to describe the architectural view and used FSP (a process algebraic language) to model the behavioral view of an adaptive program. They used *property automata* to specify the properties to be held by the adaptive program and used LTSA to verify these properties. A quiescent state in their approach refers to the state in which the component to be changed is in a passive state, and all communication with the component initiated by other components have completed. Their work highlighted the importance of identifying the states in which adaptation may be correctly performed, and it provided insight into how to use model checking to verify adaptation correctness. Allen *et al* [9] integrated the specifications for both the architectural and the behavioral aspects of dynamic programs using the Wright ADL. They use two separate component specifications to describe the behavior of a component before and after adaptation and encapsulate the dynamic changes in the "glue" specification of a connector, achieving separation of concerns. The Wright specification can be converted into the process algebraic language, CSP [29], which can then be statically ver-

ified. Canal *et al* [11] used LEDA, an ADL that supports inheritance and dynamic reconfiguration, to specify dynamic programs. LEDA is based on the π-calculus, a simple, but powerful process algebra. The richer, more expressive nature of the π-calculus enables modelers to express dynamic component connections more easily when compared to CSP-based approaches. It is also possible to derive prototypes and interfaces from the specification automatically.

Below are some of the key differences when comparing the above approaches to the one presented in this paper. (1) The above approaches do not take into consideration the impact of adaptation mechanisms when defining quiescent states, nor do they evaluate the quiescent states in the contex of global, high-level requirements. (2) None of the above approaches support state transfer, which makes it necessary for the programs to wait or even be blocked until a quiescent state is reached. (3) The specifications for adaptive behavior are still entangled with the specifications for non-adaptive behavior in the sense that the quiescent states for adaptations are specified as part of the source specifications, instead of as part of the adaptation specifications. (4) The adaptive actions in all three approaches are simple actions. None of the above approaches has considered the coordination among concurrent adaptive actions. (5) The above techniques are specific to the type of formalism being used (process algebra), making them potentially difficult to be generalized to other types of formalisms.

Many efforts in the Software Engineering and Network Systems (SENS) Laboratory at Michigan State University have explicitly addressed the correctness issue of software adaptation. As part of the RAPIDware project [30], we recently introduced a general, efficient, algorithm [31] that manages a safe dynamic adaptation process and handles failures that might occur during the adaptation with a rollback mechanism. Also, as part of the RAPIDware project, Kulkarni *et al.* [32] proposed an approach to safely compose distributed fault-tolerance components at run time. In more recent work [33], they introduced a transitional-invariant lattice technique. They used theorem proving techniques to show that during and after an adaptation, the adaptive program is always in correct states in terms of satisfying the transitional-invariants.

Other dynamic adaptation techniques have also explicitly addressed correctness issues. C2 [7] is an architectural style proposed by Taylor *et al.* They also developed ArchStudio [6], a management tool for dynamic C2 style software evolution. Cactus [34] is a system for constructing highly configurable distributed services and protocols. In Cactus, a host is organized hierarchically into layers where each layer includes many adaptive components. Chen *et al.* [34] proposed a *graceful adaptation protocol* that allows adaptations to be coordinated across hosts transparently to the application. Appavoo *et al.* [35] proposed a *hot-swapping* technique, which refers to run-time object replacement. In their approach, a *quiescent state* of an object is the state in which no other process is currently using any function of the object.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a model-driven software development process for dynamic programs. The process focuses on behavioral modeling. We described the relationships among three different levels of system abstraction: high-level requirements, state-based models for adaptive components, and low-level implementation. We introduced a technique for creating adaptive models based on high-level requirements, as well as verifying and validating the adaptive models. Furthermore, we described how to use these models to generate executable adaptive programs.

Compared to other process algebra-based approaches [9, 10, 11], our approach has several advantages. First, we define the quiescent states in the context of the source/target behavior, adaptation state transformation function, and the global invariants. Second, our approach is capable of expressing adaptations with state transformation, allowing more flexible adaptations. In our approach, we analyze the specification for different adaptation semantics, including overlap adaptation, which specifically deals with the coordinations among parallel adaptations. Third, we specify separately the program behavior before, during, and after adaptation. Also, as we believe that quiescent states should be a feature of adaptation, rather than a feature of the source program, we allow the adaptation specification to select the source states from which to adapt and to which target states to adapt. Finally, our approach is fairly general and can be instantiated in terms of several different formalisms.

The running adaptive example had been originally developed in our lab without the proposed approach. After applying our approach to the example, we found the problem to be significantly better understood by the developers and the design to be clearer. Our approach significantly reduced the developer's burden by separating different concerns and utilizing automated model construction and analysis tools, thereby improving both the development time and reliability.

Several issues require further investigations. (1) An adaptive program with N different steady-state behaviors potentially has N^2 different ways of adaptation. Our approach requires each adaptation to be individually modeled and verified, which may be expensive when N is large. This problem is shared by the other three process algebra-based approaches. We are investigating approaches to reduce the verification cost by reuse and modularization [36, 37, 38]. (2) Unlike the other three process algebra-based approaches, our approach only focuses on the behavioral aspect of adaptive programs. We also realize the importance of expressing adaptations at the architectural level. We believe an integration of our approach with an appropriate ADL representation will provide a more comprehensive solution to the structural and behavioral development of dynamic adaptive programs. (3) The features of our approach are supported by different tool suites. We would like to build an integrated environment that coordinates different tools to support our approach.

Acknowledgements

The authors gratefully acknowledge members of the Software Engineering and Network Systems Laboratory at Michigan State University who contributed to this work. We especially appreciate the feedback on this paper from Philip K. McKinley, Min Deng, Kurt Stirewalt, and Ali Ebnenasir. In addition, we appreciate Zhinan Zhou's efforts in providing the motivating example and helping to validate our technique. This work has been supported in part by NSF grants EIA-0000433, EIA-0130724, ITR-0313142, and CCR-9901017, and the Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744.

9. 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] J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger, "A survey of self management in dynamic software architecture specifications," in *Proc. of the ACM SIGSOFT International Workshop on Self-Managed Systems (WOSS'04)*, (Newport Beach, California), pp. 28–33, October/November 2004.

[3] D. L. Métayer, "Software architecture styles as graph grammars," in *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 15–23, ACM Press, 1996.

[4] G. Taentzer, M. Goedicke, and T. Meyer, "Dynamic change management by distributed graph transformation: Towards configurable distributed systems," in *Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pp. 179–193, Springer-Verlag, 2000.

[5] D. Hirsch, P. Inverardi, and U. Montanari, "Graph grammars and constraint solving for software architecture styles," in *Proceedings of the third international workshop on Software architecture*, pp. 69–72, ACM Press, 1998.

[6] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proceedings of the 20th international conference on Software engineering*, pp. 177–186, IEEE Computer Society, 1998.

[7] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins, "A component- and message-based architectural style for GUI software," in *Proceedings of the 17th international conference on Software engineering*, pp. 295–304, ACM Press, 1995.

[8] J. Kramer, J. Magee, and M. Sloman, "Configuring distributed systems," in *Proceedings of the 5th workshop on ACM SIGOPS European workshop*, pp. 1–5, ACM Press, 1992.

[9] R. Allen, R. Douence, and D. Garlan, "Specifying and analyzing dynamic software architectures," in *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, (Lisbon, Portugal), March 1998.

[10] J. Kramer and J. Magee, "Analysing dynamic change in software architectures: a case study," in *Proc. of 4th IEEE international conference on configuratble distributed systems*, (Annapolis), May 1998.

[11] C. Canal, E. Pimentel, and J. M. Troya, "Specification and refinement of dynamic software architectures," in *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pp. 107–126, Kluwer, B.V., 1999.

[12] Z. Zhou, P. K. McKinley, and S. M. Sadjadi, "On quality-of-service and energy consumption tradeoffs in fec-encoded wireless audio streaming," in *Proceedings of the 12th IEEE International Workshop on Quality of Service (IWQoS 2004)*, (Montreal, Canada), June 2004. best paper award.

[13] C. A. Petri, *Kommunikation mit A utomaten*. PhD thesis, Sehriften des Institutes fiir instrumentelle Mathematik, Bonn, Germany, 1962.

[14] J. L. Peterson, "Petri nets," *ACM Comput. Surv.*, vol. 9, no. 3, pp. 223–252, 1977.

[15] M. Mäkelä, "Maria: Modular reachability analyser for algebraic system nets," in *ICATPN '02: Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, (London, UK), pp. 434–444, Springer-Verlag, 2002.

[16] O. Kummer and F. Wienberg, "Renew - the reference net workshop," in *In Tool Demonstrations, 21st International Conference on Application and Theory of Petri Nets*, (Aarhus, Denmark), pp. 28–30, 2000.

[17] J.-C. Bolot and A. Vega-Garcia, "Control mechanisms for packet audio in the internet," in *Proceedings of IEEE INFO-COM96*, (San Francisco, California), pp. 232–239, 1996.

[18] O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 97–107, ACM Press, 1985.

[19] D. M. Berry, B. H. Cheng, and J. Zhang, "The four levels of requirements engineering for and in dynamic adaptive systems," in *Proc. of 11th International Workshop on Requirements Engineering: Foundation for Software Quality*, (Porto, Portugal), June 2005.

[20] J. Zhang and B. H. Cheng, "Specifying adaptation semantics," in *Proceedings of ICSE 2005 Workshop on Architecting Dependable Systems*, (St. Louis, Missouri), May 2005.

[21] A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu, "Towards requirements-driven autonomic systems design," in *Proceedings of ICSE 2005 Workshop on Design and Evolution of Autonomic Application Software*, (St. Louis, Missouri), May 2005.

[22] A. van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, p. 249, IEEE Computer Society, 2001.

[23] M. B. Dwyer and C. S. Pasareanu, "Filter-based model checking of partial systems," in *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 189–202, ACM Press, 1998.

[24] J. Zhang and B. H. Cheng, "Model-based development of dynamically adaptive software," Tech. Rep. MSU-CSE-05-24, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, September 2005.

[25] H. Bowman and S. J. Thompson, "A tableaux method for Interval Temporal Logic with projection," in *TABLEAUX'98, International Conference on Analytic Tableaux and Related Methods*, no. 1397 in Lecture Notes in AI, pp. 108–123, Springer-Verlag, May 1998.

[26] A. Pretschner, "Model-based testing," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, (New York, NY, USA), pp. 722–723, ACM Press, 2005.

[27] B. Potter, J. Sinclair, and D. Till, *An introduction to Formal Specification and Z*. Prentice Hall International (UK) Ltd, 1991.

[28] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*. Addison-Wesley, 1999.

[29] C. A. R. Hoare, *Communicating sequential processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[30] "RAPIDware." http://www.cse.msu.edu/rapidware/.

[31] J. Zhang, Z. Yang, B. H. Cheng, and P. K. McKinley, "Adding safeness to dynamic adaptation techniques," in *Proceedings of ICSE 2004 Workshop on Architecting Dependable Systems*, (Edinburgh, Scotland, UK), May 2004.

[32] S. S. Kulkarni, K. N. Biyani, and U. Arumugam, "Composing distributed fault-tolerance components," in *Proccedings of the International Conference on Dependable Systems and Networks (DSN), Supplemental Volume, Workshop on Principles of Dependable Systems*, pp. W127–W136, June 2003.

[33] S. Kulkarni and K. Biyani, "Correctness of component-based adaptation," in *Proceedings of International Symposium on Component-based Software Engineering*, May 2004.

[34] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting, "Constructing adaptive software in distributed systems," in *Proc. of the 21st International Conference on Distributed Computing Systems*, (Mesa, AZ), April 16 - 19 2001.

[35] J. Appavoo, K. Hui, C. A. N. Soules, *et al.*, "Enabling autonomic behavior in systems software with hot swapping," *IBM System Journal*, vol. 42, no. 1, p. 60, 2003.

[36] S. Krishnamurthi, K. Fisler, and M. Greenberg, "Verifying aspect advice modularly," in *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, (New York, NY, USA), pp. 137–146, ACM Press, 2004.

[37] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. Sanvido, "Extreme model checking," *Verification: Theory and Practice, Lecture Notes in Computer Science 2772, Springer-Verlag*, pp. 332–358, 2004.

[38] M. Mäkelä, "Model checking safety properties in modular high-level nets," in *Application and Theory of Petri Nets 2003, 24th International Conference, ICATPN 2003 (volume 2679 of Lecture Notes in Computer Science)*, (Eindhoven, The Netherlands), pp. 201–220, Springer-Verlag, June 2003.