This is to certify that the
dissertation entitled

A FORMAL APPROACH TO PROVIDING ASSURANCE TO DYNAMICALLY ADAPTIVE SOFTWARE

presented by

JI ZHANG

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Computer Science and Engineering

Betty H.C. Cheng
Major Professor's Signature

5/1/2007
Date

DATE DUE	DATE DUE	DATE DUE

A FORMAL APPROACH TO PROVIDING ASSURANCE TO DYNAMICALLY ADAPTIVE SOFTWARE

By

Ji Zhang

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science

2007

ABSTRACT

A FORMAL APPROACH TO PROVIDING ASSURANCE

TO DYNAMICALLY ADAPTIVE SOFTWARE

By

Ji Zhang

Increasingly, software must adapt its behavior in response to the changes in its run-time environment and user requirements in order to upgrade services, to harden security, or to improve performance. In order for adaptive software to be used in safety critical and mission critical systems, they must be trusted. Adaptive software assurance must be addressed at different stages of the software development process, including the requirements analysis phase, the design phase, and the implementation phase. An adaptation-oriented systematic software development process that applies formal methods throughout the process can be used to provide assurance to adaptive systems. This dissertation introduces a number of specification languages, modeling techniques, and model checking techniques to support a systematic approach to providing assurance to adaptive software from requirements through design and implementation phases. We introduce A-LTL, an adaptation extension to LTL, and a goal-based requirements analysis technique to formally specify adaptation requirements. We develop a model-based design technique to describe the designs that satisfy the adaptation requirements. Verification techniques are proposed to ensure that the artifacts produced in later phases conform to artifacts produced in earlier ones. Safe adaptation protocols and model checking techniques are applied to ensure that these designs are correctly followed and the requirements are satisfied in the implementation. We have applied our techniques to a number of case studies involving adaptive mobile computing applications.

To my wife and my parents and sister in China for their support and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Increasingly, computer software must adapt to changing conditions in both the supporting computing and communication infrastructure, as well as in the surrounding physical environment [96]. To meet the needs of emerging and future adaptive systems, numerous research efforts in the past several years have addressed ways to construct adaptive software. Examples include support for adaptability in programming languages [1, 65, 114], frameworks to design context-aware applications [43, 126], dynamic architectures for reconfigurable software components and connectors [2, 92, 98, 105], adaptive middleware platforms that insulate applications from external dynamics [15, 69], and adaptable and extensible operating systems [6, 11, 39]. However, despite these advances in *mechanisms* used to build adaptive software, the full potential of dynamically adaptive software systems can be realized only if we can establish and ensure the consistency of the system during and after adaptation [150, 152].

1.1 Problem Statement

Assurance is even more important when adaptive software is used in safety and mission critical systems. For safety critical systems, errors in the software systems

may have catastrophic consequences. For mission critical systems, the cost to fix an error in the software after the mission has been launched may be exceedingly high. In order to be used in such safety and mission critical systems, the adaptive software must be trusted. Critical properties, such as *safety properties*, must be ensured in the software by the software development process.

In order for us to gain confidence in an adaptive system, assurance must be addressed at different stages of the software development process, including the requirements analysis phase, the design phase, and the implementation phase. During the requirements analysis phase, assurance for adaptation can be achieved only if the software developers fully understand the adaptation requirements for the software, which requires precise specifications of requirements. In the design phase, the requirements for the adaptive software should be satisfied in the design, and should be amenable to verification. Also, for maintainability purposes, the design of adaptive software should separate adaptive behavior from non-adaptive behavior. In the implementation phase, distributed adaptation actions should be coordinated to ensure critical constraints (e.g., dependency relationships) among components of adaptive software systems. Verification techniques may be used to verify the implementations against the requirements specifications for adaptive software. The above phases must be addressed systematically so that conditions expressed in the requirements are satisfied in the implementation.

Adaptive software is generally more difficult to specify, verify, and validate due to its high complexity [144]. Particularly, when adaptations are multi-threaded, 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 executed in a coordinated fashion.

Thesis statement: An adaptation-oriented software development process that systematically applies formal methods throughout the process can be used to gain assurance for adaptive systems.

As depicted in Figure 1.1, our approach provides assurance to the development process of adaptive software in the requirements, design, and implementation phases. In the requirements phase, formal specification languages can be used to specify adaptation requirements. We developed A-LTL [143, 146], an adaptation extension to the Linear Temporal Logic (LTL) [111] to enable the formal specification of the objectives for the software. We also developed TA-LTL [155] to specify real-time properties of adaptive systems. We applied the temporal logics to formally specify three frequently occurring adaptation semantics [143, 146]. The formal temporal logic specifications enable us to perform numerous automated analyses, such as verifying specification consistency, model checking for safety properties, etc. We introduced a goal-based requirements analysis technique [148] for analyzing adaptation requirements and constructing A-LTL and LTL specifications for adaptation. The formal specifications produced in this phase will be utilized by the design phase.

We have developed a model-based design technique [144] to describe the designs that satisfy the adaptation requirements and to mitigate the complexity in the design. The models that satisfy non-adaptive requirements are separated from those that satisfy adaptive requirements. We capture *quiescent states* (the states from which adaptation may safely start) in the design for adaptations in terms of specific adaptation contexts, including the program behavior, the requirements for the adaptation, and the adaptation mechanism. The design models can be analyzed using automated tools to determine whether they satisfy adaptation requirements. These models can then be used to generate rapid prototypes and test cases. We also introduce a technique that uses formal models to provide assurance in re-engineering

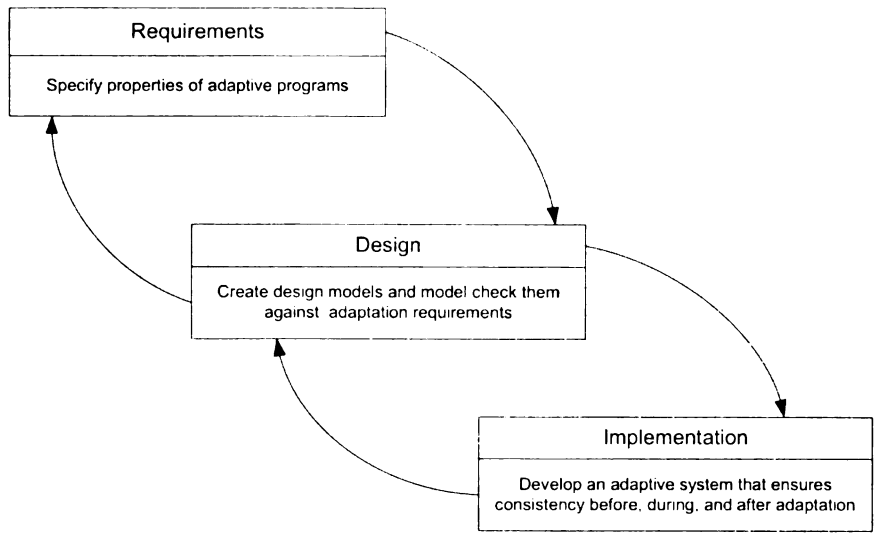


Figure 1.1: Assurance techniques in three phases

legacy software for adaptation [148].

In the implementation phase, safe adaptation protocols can be applied to ensure that adaptation designs will be followed correctly in the implementation. Our safe adaptation protocol [150, 152] ensures that safe states for an adaptation are reached before the adaptation occurs, and that the adaptation steps are performed in such a way that does not violate the dependency relationships among components of adaptive software. Our approach provides centralized management of adaptations, thereby enabling optimizations when more than one set of adaptive actions can be used to satisfy a given adaptation goal.

Critical properties in adaptive software can be verified using formal analysis techniques. We have developed a modular model checking technique AMOEBA for adaptive software [145], which not only supports the verification of A-LTL, but also significantly reduces model checking complexity for adaptive software. We also developed a run-time model checker, AMOEBA-RT, for adaptive software [147]. AMOEBA-RT uses an aspect-oriented technique [132] to insert instrumentation code in adaptive

software to collect run-time conditions. These conditions are then sent to a run-time model checking server, which verifies these conditions at run time against A-LTL/LTL properties.

1.2 Summary of Contributions

The following is a list of contributions made by the thesis.

Requirements Phase.

- We propose a finite state machine model for adaptive software [144, 145]. This model enables existing analysis techniques for traditional, non-adaptive software to be directly applied to adaptive software. Also, it enables us to exploit specific characteristics of adaptive software in order to optimize their analysis [144] (Chapter 2).

- We introduce A-LTL, the adapt operator-extended linear temporal logic. We use A-LTL to formally specify adaptation properties of adaptive software [143, 146] (Chapter 3).

- We generalize three adaptation semantics, namely the one-point adaptation, the guided adaptation, and the overlap adaptation, and formally specify the semantics in A-LTL [143, 146] (Chapter 3).

- We introduce a goal-based technique to systematically analyze the requirements for adaptive software and to generate formal requirements specifications in A-LTL [144, 149] (Chapter 4).

- We introduce TA-LTL, a real-time extension to A-LTL for specifying critical real-time constraints in adaptive software. Three types of critical properties are identified, namely safeness, liveness, and stability properties [153, 155] (Appendix A).

Design Phase.

- We propose a formal design modeling technique for adaptive software. We identify the key features of adaptive software design (i.e., quiescent states, adaptive states, and adaptive transitions) and introduce a state-based modeling approach (MASD) to capture these features with design models [144]. We also introduce rapid prototyping and model-based testing to carry these features in the models to their implementations [144] (Chapter 5).

- We propose a model-based re-engineering technique to enable adaptation in legacy software with assurance by leveraging the MASD approach [144], the metamodel-based language translation technique [97], and the aspect-oriented adaptation enabling technique [139]. We also introduce the cascade adaptation mechanism to handle state transformation from a source program to a target program in an adaptation [149] (Chapter 6).

Implementation Phase.

- We describe AMOEBA, a modular model checker that modularly verifies adaptive software against both LTL and A-LTL properties. The proposed technique reduces the model checking complexity by a factor of n, where n is the number of steady-state programs encompassed by the adaptive program [145] (Chapter 7).

- We describe a run-time model checker AMOEBA-RT that monitors the run-time conditions in adaptive software and verifies these conditions against LTL and A-LTL properties (Chapter 8).

- We introduce a safe software adaptation protocol that minimizes the adaptation cost and ensures consistencies among adaptive components. A retry/roll-back mechanism is employed to ensure the consistency of adaptive actions in the presence of failures [150, 152] (Chapter 9).

1.3 Outline of the Thesis

This thesis is presented in three parts to reflect the research investigations for the three key phases of software development: requirements, design, and implementation. Part I, comprising Chapters 1–4, discusses formal techniques for the requirements analysis phase of adaptive software development. Chapter 2 describes a finite state model for adaptive software, which serves as the foundation of the thesis. Chapter 3 introduces the A-LTL specification language and three frequently occurring adaptation semantics in adaptive software. Chapter 4 presents a goal-based requirements analysis approach to derive adaptation requirements specifications in A-LTL and LTL. Part II, comprising Chapters 5–6, introduces assurance techniques for the design phase of adaptive software development. Chapter 5 describes the model-based design technique for adaptive software. Chapter 6 extends the model-based design technique to provide assurance in re-engineering legacy software for adaptation. Part III, comprising Chapters 7–9, introduces assurance mechanisms and analysis techniques for the implementation phase of adaptive software development. Chapter 7 presents the AMOEBA modular model checking technique, and the run-time adaptive software verification technique AMOEBA-RT is described in Chapter 8. Chapter 9 describes a safe adaptation protocol for distributed adaptive software systems. Finally, Chapter 10 presents concluding remarks and outlines potential future investigations. Several appendices elaborate details of the work. Appendix A describes TA-LTL, a real-time extension of A-LTL for specifying real-time constraints, including safety, liveness, and stability properties in adaptive software. Appendix B provides supporting material for Chapter 5. Appendix C provides supporting material for Chapter 6.

Part I

Requirements Analysis for Adaptive Software

Chapter 2

A Formal Model For Adaptive Software

This chapter describes a finite state formal model for adaptive software. This model serves as the foundation of our specification and analysis techniques throughout the thesis. This chapter is organized as follows. Section 2.1 gives background information on a finite state machine model for general software. Section 2.2 describes the formal finite state model for adaptive software and two composition operations. Section 2.3 outlines the architecture of autonomic computing system. Related work is described in Section 2.4. Section 2.5 discuss possible extensions to the formal model introduced in this chapter.

2.1 Background: Program Kripke Structure

In this section, we introduce a finite state model for general software. A non-terminating program can be considered an ω-automaton [133], i.e., a finite state automaton that accepts ω-words.[1] A program state can be represented by the truth value of a set of propositions in the state called *atomic propositions*, denoted AP.

[1] An ω-word can be considered as a sequence of infinite states.

Emerson *et al* [37] defined a program as a *Kripke structure* $M = (S, P, L)$, where

- S represents a finite set of program states.
- $P : S \leftrightarrow S$ represents nondeterministic program transitions. Relation P is total, meaning that $\forall s \in S, \exists s' \in S$ such that $(s, s') \in P$.
- $L : S \rightarrow 2^{AP}$ maps each state to a set of atomic propositions that are true in the state.

A *computation* of a program is an infinite sequence of states $\sigma = s_0, s_1, ...$ such that for each $i \geq 0$, $(s_i, s_{i+1}) \in P$.

2.2 Finite State Model For Adaptive Software

In general, a *program* exhibits certain behavior and operates in a certain *domain*, where a domain is defined to be the input space of the program [10]. A *dynamically adaptive program* operates in different domains, changes its behavior at run time in response to changes of the domains. In this thesis, we take a general view of programs, i.e., an adaptive program is a program whose state space can be separated into a number of disjoint regions, each of which exhibits a different *steady-state* behavior [2], and operates in a different domain. The state space exhibiting each different kind of *steady-state* [2] behavior is a *steady-state program*, or briefly a *program*. The states and transitions connecting one steady-state program to another are *adaptation sets*.

An adaptive program usually contains multiple steady-state programs and multiple adaptation sets connecting these programs. We term an adaptive program comprising n different steady-state programs an *n-plex adaptive program*. Initially, we simplify our discussion 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* (the program from which the adaptation starts), a *target program* (the program in which the adaptation ends), and an adaptation set connecting the source program to the target program. Figure 2.1 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. Accordingly, a general n-plex adaptive program can be considered as the union of one or more simple adaptive programs.

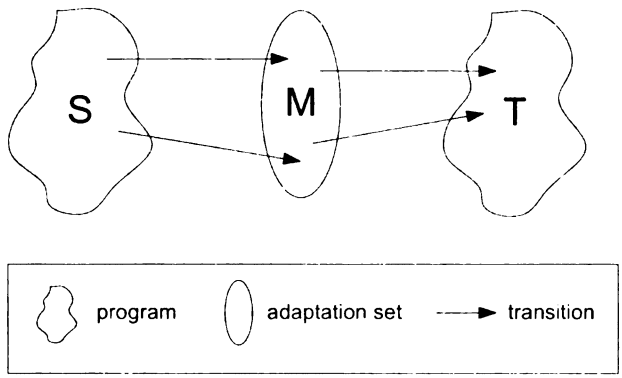


Figure 2.1: A simple adaptive program

2.2.1 Formal Definition

The *Finite State Machine* (FSM) model for adaptive software is formally defined as follows: Given a set of atomic propositions AP, a *finite-state machine* (FSM) is a tuple $M = (S, S_0, T, L)$, where

- S is a set of states;
- the initial state set $S_0 \subseteq S$ is a subset of the states;
- transitions $T : S \times S$ is a set of state pairs, where $(s, t) \in T$ represents that there is an arc from s (the predecessor) to t (the successor);
- the function $L : S \rightarrow 2^{AP}$ labels each state s with a set of atomic propositions that are evaluated *true* in s.

We represent the states, the initial states, the transitions, and the labels of a given program model M with $S(M)$, $S_0(M)$, $T(M)$, and $L(M)$ respectively. The states in M that do not have successor states are *deadlock states*, denoted $D(M)$. We can eliminate deadlock states in an FSM by introducing a self-loop to each deadlock state. An FSM is an *Extended FSM* (EFSM) if it does not contain a deadlock state.

We formally model an *n-plex adaptive program* as an EFSM that contains n steady-state programs $P_1, P_2, \cdots, P_n$, each of which is an EFSM. Each steady-state program represents a different program behavior for a run-time execution domain. We require that these steady-state programs be state disjunct, i.e., no two steady-state programs may share any state:

$$P_1 \neq P_2 \Rightarrow S(P_1) \cap S(P_2) = \varnothing. \tag{2.1}$$

The adaptation from P_i (the *source program*) to P_j (the *target program*) is modeled by an *adaptation set* $A_{i,j}$. An adaptation set contains the intermediate states and transitions connecting one steady-state program to another, representing a collaborative adaptation procedure, such as the insertion and removal of filters [150, 152].

An adaptation set is formally defined as an FSM with the following features:

- All initial states are states in the source program:

  $$S_0(A_{i,j}) \subseteq S(P_i).$$

- All deadlock states are states in the target program:

  $$D(A_{i,j}) \subseteq S(P_j).$$

- No transition should return from the adaptation set to the source program:

  $$\forall s, t : S(A_{i,j}).\ (s, t) \in T(A_{i,j}) \Rightarrow t \notin S(P_i).$$

- No transition should return from the target program to the adaptation set:

  $$\forall s, t : S(A_{i,j}).\ (s, t) \in T(A_{i,j}) \Rightarrow s \notin S(P_j).$$

- There should be no cycles in the adaptation set. This condition ensures the adap-

tation integrity constraint [144], i.e., the adaptation should finally reach a state of the target program.

- No two adaptation sets may share the same states other than those in the target and source programs.

Figure 2.2 shows the metamodel for the FSM representation of adaptive programs. An adaptive program aggregates a set of atomic propositions, a set of steady-state programs, and a set of adaptation sets. Each steady-state program or adaptation set aggregates a set of states and transitions. A state has a state ID, and satisfies a subset of the atomic propositions in the adaptive program. A transition is a pair of incoming and outgoing state IDs.

2.2.2 Compositions of Adaptive Programs

We define the *sequential composition* of two programs $comp(P_i, P_j)$ to be a program with all the states and transitions in P_i and P_j, and with initial states coming from P_i:

$$\begin{aligned} comp(P_i, P_j) &= (S, S_0, T, L), \text{ where} \\ S &= S(P_i) \cup S(P_j),\ S_0 = S_0(P_i), \\ T &= T(P_i) \cup T(P_j), \text{ and } L = L(P_i) \cup L(P_j). \end{aligned}$$

The *comp* operation can be recursively extended to accept a list of programs:

$$comp(P_{i_1}, P_{i_2}, \cdots, P_{i_n}) = comp(\cdots comp(P_{i_1}, P_{i_2}), \cdots, P_{i_n}).$$

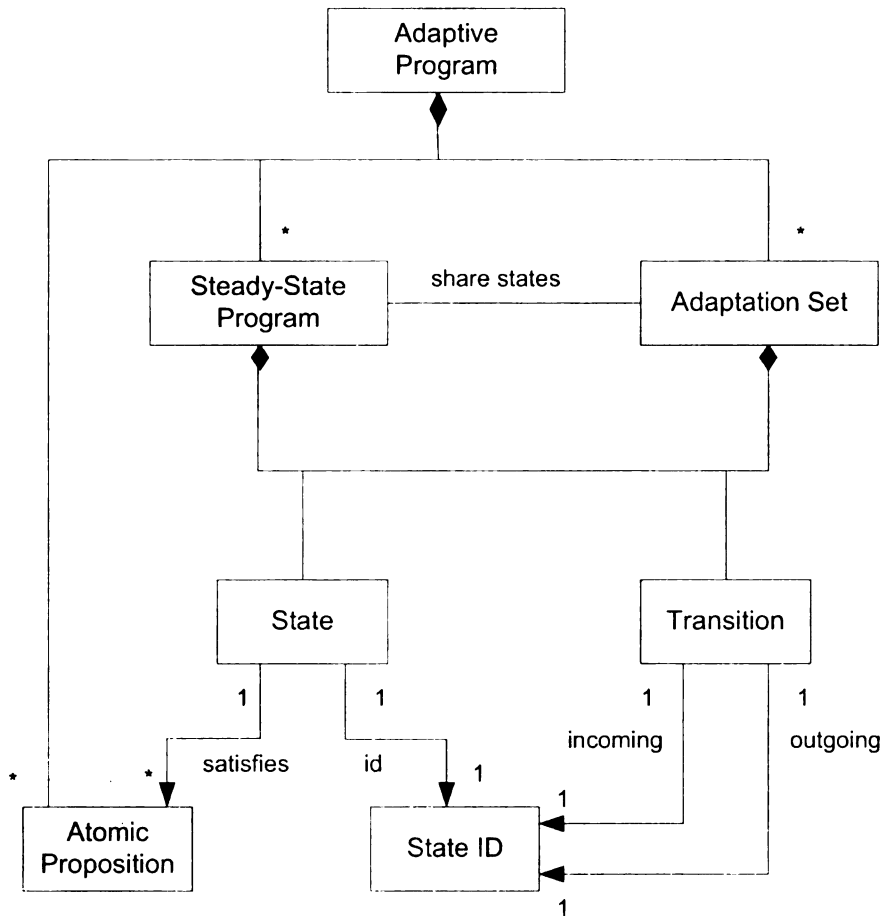


Figure 2.2: Metamodel for adaptive program

Similarly, we define the *union* of two programs $union(P_i, P_j)$ to be a program with all the states, transitions, and initial states from both P_i and P_j:

$$\begin{aligned} union(P_i, P_j) &= (S, S_0, T, L), \text{where} \\ S &= S(P_i) \cup S(P_j),\ S_0 = S_0(P_i) \cup S_0(P_j). \\ T &= T(P_i) \cup T(P_j),\ L = L(P_i) \cup L(P_j). \end{aligned}$$

The *union* operation can be extended to accept a list of programs:

$$union(P_{i_1}, P_{i_2}, \cdots, P_{i_n}) = union(\cdots union(P_{i_1}, P_{i_2}), \cdots, P_{i_n}).$$

A *simple adaptive program* $SA_{i,j}$ from P_i to P_j includes the source program P_i, the target program P_j, and the adaptation set $A_{i,j}$ that comprises the intermediate states and transitions connecting P_i to P_j. Formally, we define the simple adaptive program from P_i to P_j as the composition

$$SA_{i,j} = comp(P_i, A_{i,j}, P_j).$$

An *n-plex adaptive program* M contains the union of all the states, transitions, and initials states of n steady-state programs and the corresponding adaptation sets. Formally:

$$M = comp(union(P_1, \cdots P_n), union(A_{1,2}, A_{1,3}, \cdots A_{n,n-1})).$$

An *execution* of an n-plex adaptive program M is an infinite state sequence $s_0, s_1, s_2, \cdots$ such that $s_i \in S(M)$, $(s_i, s_{i+1}) \in T(M)$, and $s_0 \in S_0(M)$ (for all $i \geq 0$). A *non-adaptive execution* is an execution $s_0, s_1, s_2, \cdots$, such that all its states are within one program: $s_i \in P_j$, for all s_i and some P_j. An *adaptive execution* is any execution that is not non-adaptive. An adaptive execution goes through one or more adaptation sets, and two or more steady-state programs.

2.3 Autonomic Computing System

Kephart *et al* [66] proposed *autonomic computing* as an approach to manage the exploding complexity of computing systems. In an autonomic computing system, software interacts with its run-time environment, and adjusts its own behavior in order to achieve self-managing, self-healing, self-optimizing, self-protecting, etc [24, 33, 50, 66]. A general autonomic system is depicted in Figure 2.3 where the central part of the system is an adaptive program interacting with its run-time environment. The adaptive program includes an adapt-ready program [139], a number of monitors, a decision maker, and an adaptation coordinator. An adapt-ready program is a program whose behavior can be altered at run time to exhibit a number of different steady-state behaviors in response to adaptation requests. The monitors collect run-time internal data in the adapt-ready program (e.g., variable values) and external data in the environment (e.g., network bandwidth), and evaluate a set of predefined conditions (e.g., lossrate $<$ 0.2) upon these data. When a condition is met, the monitors sends the condition to the decision maker. The decision maker, comprising a set of decision making rules, which are either predefined or re-loadable at run time, makes adaptation decisions based on the conditions received from the monitors and sends these adaptation decisions to the adaptation coordinator. The adaptation coordinator decomposes an adaptation decision into a sequence of adaptation requests, which are then sent to the adapt-ready program to alter its behavior. The assurance addressed by this thesis cross cuts all the components in the adaptive program.

2.4 Related Work

Adaptive software has been studied by researchers for more than a decade. Numerous researchers have proposed ways to formally specify dynamically adaptive programs [18].

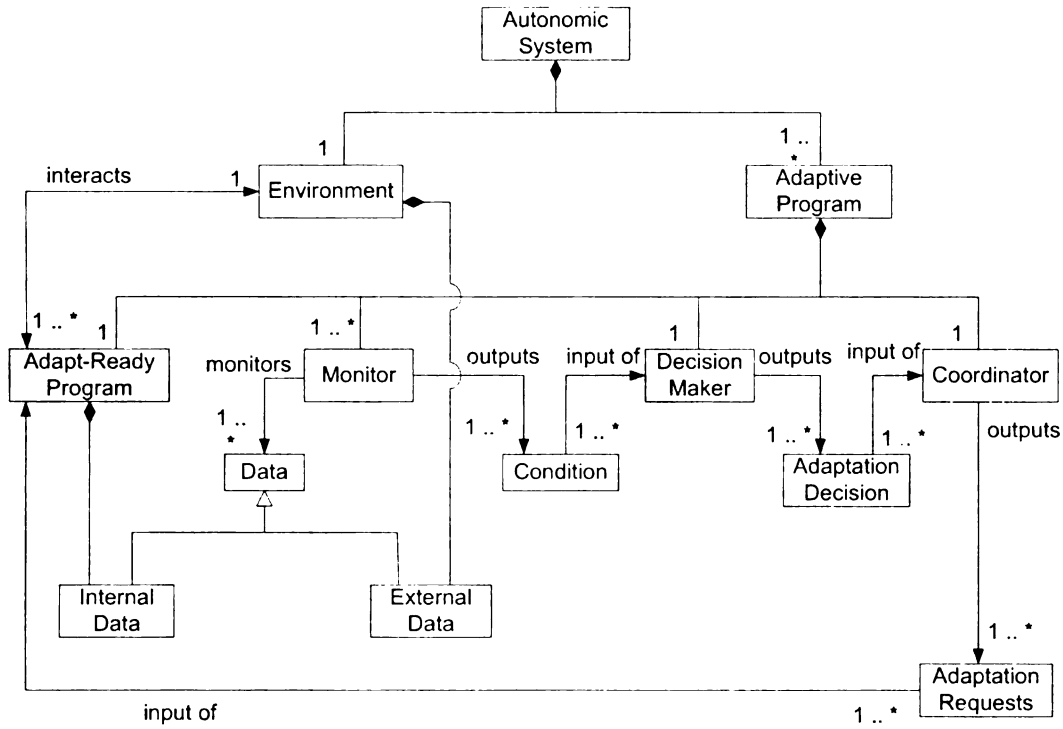


Figure 2.3: General architecture for autonomic system

Graph-based approaches have been proposed to model the dynamic architectures of adaptive programs as graph transformations [57, 99, 128]. In the Hypergraph approach [57], for example, an adaptive program is defined as a hypergraph $G = (N, E, L)$, where N is a set of nodes representing communication ports, E is a set of edges representing components, and L is a set of labeling functions for nodes and edges. A node label includes the name of the port; an edge label includes the name of the component and its current status. The components can be connected to each other through ports. A software architecture style is a set of software architectures with similar structures, described by a *hyper-edge context-free grammar*. A graph transformation production rule has the format $L \rightarrow R$, where L describes the graph to be rewritten, and R describes the graph to be generated. They defined three types of rules: Construction rules are used to construct the initial structure of the software;

dynamic evolution rules are used to transform the software structure dynamically; and communication rules describe the communication among components in the software.

Architecture Description Language (ADL)-based approaches are also introduced to model adaptive software [73, 106, 131]. ADL approaches are similar to the graph-based approaches in the sense that a software architecture is represented by components connected with connectors. The major difference between these two categories is that in ADL-based approaches, the structural changes are defined as disconnections and reconnections of the connectors, rather than graph transformation production rules. For example, Darwin [91, 92] is an ADL used to specify software configurations/reconfigurations. In Darwin, components are represented as boxes. Each component declares its *requirement* ports (denoted as empty circles) representing the services required by the components and *provision* ports (denoted as full circles) representing the services provided by the component. Components are connected with one another by binding provision ports with requirement ports. Dynamic structure is achieved by the definitions of bindings between ports of component "types" rather than component instances. Thus, the ports of dynamically instantiated components can be identified and bound to one another. Darwin supports two techniques for dynamic software bindings: direct dynamic instantiation and lazy instantiation. Direct dynamic instantiation allows port references to be passed as messages among components. Lazy instantiation allows a component to delay its instantiation until another component tries to use a service that it provides. The formal semantics of the Darwin architecture is defined in π-calculus.

Both the graph-based approaches and the ADL-based approaches focus on structural changes of adaptive programs. In contrast, our approach focuses on behavioral changes of adaptive programs.

2.5 Discussion

Our work uses finite state machines to model adaptive software. This strategy enables existing analysis techniques for traditional non-adaptive software to be directly applied to adaptive software. Also, it enables us to exploit certain characteristics of adaptive software in order to optimize the analysis for adaptive software.

The finite state machine models for adaptive programs introduced in this chapter are abstract representations of adaptive software implementations. Existing tools, including Bandera [28], FLAVERS [26], Borgor [117], etc., process programs in high-level programming languages, such as Ada, Java, and C++, and generate finite state models. We envision that these techniques can be leveraged to generate adaptive program models described in this chapter.

In order to specify real-time programs, researchers have also extended the models with real-time properties [3, 30, 37, 48, 88, 89]. The finite state model for adaptive programs introduced in this chapter can also be extended with real-time information, including those introduced in the *timed state graph* [3], to represent real-time adaptive programs.

Chapter 3

Temporal Logic for Specifying Adaptive Programs

This chapter describes the *Adapt operator-extended Linear Temporal Logic* (A-LTL) [143, 146], an extension to LTL [111], to formally specify adaptation from one steady-state program to another. We introduce three basic adaptation semantics and use A-LTL to formally specify these semantics. These basic adaptation semantics can be composed to derive more complex adaptation semantics. This chapter is organized as follows. Section 3.1 gives background information on the Linear Temporal Logic (LTL). Section 3.2 describes the syntax and the semantics of A-LTL and Section 3.3 introduces three commonly occurring adaptation semantics specified in A-LTL. In Section 3.4 we introduce two composition techniques to construct complex A-LTL specifications from simple ones. We demonstrate the A-LTL specification using an adaptive audio streaming example in Section 3.5. Related work is described in Section 3.6. Section 3.7 summarizes this chapter and discusses possible extensions to A-LTL.

3.1 Background: Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL), first proposed by Pnueli [111], is an extension to the propositional logic. It extends propositional logic by introducing four basic temporal operators: the *existential* operator $\Diamond$, the *global* operator $\Box$, the *until* operator $\mathcal{U}$, and the *next* operator $\bigcirc$. An LTL formula is evaluated upon a sequence of infinite states, where in each state, a propositional sub-formula is either *true* or *false*. Informally, given formulae ϕ and ψ, $\Diamond\phi$ (read as eventually ϕ) means that ϕ is eventually *true*; $\Box\phi$ (read as always ϕ) means that ϕ is always *true*; $\phi\,\mathcal{U}\,\psi$ (read as ϕ until ψ) means that ϕ is *true* until ψ is *true*; $\bigcirc\phi$ (read as next ϕ) means that ϕ is *true* in the next state. LTL has been adopted by the formal methods community to express temporal properties of software, including *safety* properties and *liveness* properties [81]. A safety property (usually having the form of "$\Box\neg\phi$") states that "*nothing bad will happen*". A liveness property (usually having the form of "$\Box\Diamond\phi$") states that "*something good will eventually happen*".

3.1.1 Syntax

The syntax of LTL is defined as follows:

1. Each atomic proposition is a formula;
2. if ϕ and ψ are formulae, then $\neg\phi$, $\phi\wedge\psi$, $\phi\,\mathcal{U}\,\psi$, $\bigcirc\phi$ are formulae.

Other operators are introduced as abbreviations of the above operators: $\Diamond\phi \equiv$ *true* $\mathcal{U}\,\phi$ (i.e., "finally ϕ" is equivalent to "*true* until ϕ"), and $\Box\phi \equiv \neg\Diamond\neg\phi$ (i.e., "always ϕ" is equivalent to "not finally not ϕ").

3.1.2 Semantics

Let AP be an underlying set of atomic propositions, the semantics of LTL is defined with respect to a linear Kripke structure $M = (S, x, L)$, where

1. S is a finite set of states;
2. $x : \mathbb{N} \rightarrow S$ is an infinite sequence of states representing an infinite linear timeline;
3. $L : S \rightarrow 2^{AP}$ is a labeling function of each state representing the atomic propositions evaluated to be true in the state.

$M, x \models \phi$ indicates that the LTL formula ϕ is true under the given structure M and timeline x, or briefly $x \models \phi$ when M is understood. The symbol x^i represents the ith suffix of x, i.e., if $x = s_0, s_1, s_2, \cdots$, then $x^i = s_i, s_{i+1}, s_{i+2}, \cdots$. The semantics of LTL is defined as follows:

1. $x \models \phi$ if and only if $\phi \in L(s_0)$, for any atomic proposition ϕ;
2. $x \models \phi \wedge \psi$ if and only if $x \models \phi$ and $x \models \psi$;
3. $x \models \phi \mathcal{U} \psi$ if and only if $\exists j$ such that $x^j \models \psi$, $\forall k (0 \leq k < j$, and $(x^k \models \phi)$;
4. $x \models \bigcirc \phi$ if and only if $x^1 \models \phi$.

An LTL formula ϕ is *satisfiable*, if and only if there exists a Kripke structure $M = (S, x, L)$, such that $M, x \models \phi$, in which case, we say M defines a *model* of ϕ. An LTL ϕ is *valid*, denoted as $\models \phi$, if and only if for all Kripke structures $M = (S, x, L)$, we have $M, x \models \phi$.

3.2 The Adapt-Operator Extended LTL (A-LTL)

In this section, we introduce the Adapt-operator extended LTL (A-LTL), a temporal logic for the specification of adaptation behavior.

3.2.1 Syntax

To specify adaptation behavior, we introduce A-LTL by extending LTL [111] with the *adapt* operator ("$\stackrel{\Omega}{\rightarrow}$") [143, 146]. The A-LTL formula "$\phi \stackrel{\Omega}{\rightarrow} \psi$" is read as "$\phi$ adapts to ψ with adaptation constraint Ω", where ϕ, ψ, and Ω are three temporal logic formulae named the *source specification*, the *target specification*, and the *adaptation constraint* of the formula, respectively. Informally, a program satisfying "$\phi \stackrel{\Omega}{\rightarrow} \psi$" means that the program initially satisfies ϕ. In a certain state A, it stops being constrained by ϕ, and in the next state B, it starts to satisfy ψ, and the two-state sequence (A, B) satisfies Ω. We use $\phi \stackrel{\Omega}{\rightarrow} \psi$ to specify the adaptation from a steady-state program that satisfies ϕ to another that satisfies ψ. Formally, we define A-LTL as follows:

- If ϕ is an LTL formula, then ϕ is also an A-LTL formula;
- if ϕ and ψ are both A-LTL formulae, and Ω is an LTL formula, then $\xi \equiv \phi \stackrel{\Omega}{\rightarrow} \psi$ is an A-LTL formula;
- if ϕ and ψ are both A-LTL formulae, then $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\Box\phi$, $\Diamond\phi$, and $\phi \mathcal{U} \psi$ are all A-LTL formulae.

3.2.2 Semantics

We define A-LTL semantics over both finite state sequences (denoted by "$\models_{fin}$") and infinite sequences (denoted by "$\models_{inf}$", or briefly, $\models$).

- Operators ($\Rightarrow$, $\wedge$, $\vee$, $\Box$, $\Diamond$, $\mathcal{U}$, $\neg$, etc.) are defined similarly as those in LTL.
- If σ is an infinite state sequence and ϕ is an LTL formula, then σ satisfies ϕ in A-LTL, if and only if σ satisfies ϕ in LTL. Formally, $\sigma \models_{inf} \phi \Leftrightarrow \sigma \models \phi$ holds in LTL.

- If σ is a finite state sequence and ϕ is an A-LTL formula, then $\sigma \models_{fin} \phi \Leftrightarrow \sigma' \models_{inf} \phi$, where σ' is the infinite state sequence constructed by repeating the last state of σ.

- $\sigma \models_{inf} \phi \xrightarrow{\Omega} \psi$, if and only if there exist a finite state sequence $\sigma' = (s_0, s_1, \cdots, s_k)$ and an infinite state sequence $\sigma'' = (s_{k+1}, s_{k+2}, \cdots)$, such that $\sigma = \sigma' \frown \sigma''$, $\sigma' \models_{fin} \phi$, $\sigma'' \models_{inf} \psi$, and $(s_k, s_{k+1}) \models_{fin} \Omega$, where ϕ, ψ, and Ω are A-LTL formulae, and the $\frown$ is the sequence concatenation operator.

  Informally, a sequence satisfying $\phi \xrightarrow{\Omega} \psi$ can be considered the concatenation of two subsequences, where the first subsequence satisfies ϕ, the second subsequence satisfies ψ, and the two states connecting the two subsequences satisfy Ω.

For convenience, we define *empty* to be the formula that is true if and only if it is evaluated upon single state sequences, i.e., *deadlock states*: $empty \equiv \neg\bigcirc true$ [17]. Also, we write $\phi \rightarrow \psi$ to represent $\phi \xrightarrow{\Omega} \psi$ when $\Omega \equiv true$.

3.2.3 Expressiveness

We now show that A-LTL and LTL are equivalent in their expressive power. An *ω-automaton* is a finite state automaton that accepts ω-words [133]. Two most popular forms of ω-automata are Büchi automata and Muller automata [133]. It has been shown that Büchi automata and Muller automata are equivalent [133]. A *counter* [137] in an ω-automaton A is a sequence of distinct states $s_0, s1, \cdots, s_m$ with $m > 0$ and a finite word π, such that for all $i \in [0, m-1]$, there is a path labeled π in A from s_i to s_{i+1}, and from s_m to s_0. An ω-automaton is *counter-free* if it does not contain counters [137]. Wilke showed that LTL is equivalent to counter-free ω-automata and introduced an algorithm to convert a counter-free automaton to an LTL formula [137]. We establish the equivalence between LTL and A-LTL by showing that

A-LTL is also equivalent to counter-free automaton. The above statements applies to both infinite and finite words [137].

Theorem 1: *A-LTL is equivalent to counter-free automata.*

Proof 1: *We need to prove (1) that any A-LTL can be accepted by a counter-free Büchi automaton and (2) that any counter-free Büchi automaton can be expressed by an A-LTL formula.*

1. *We prove by induction on the number of nested adapt "$\stackrel{\Omega}{\rightharpoonup}$" operators in an A-LTL formula.*

   ***Initial condition**: Any A-LTL formula with 0 adapt operator is also an LTL formula, which can be accepted by a counter-free ω-automaton in ω-word domain, and can be accepted by a counter-free FSA in finite word domain.*

   ***Assumption**: Any A-LTL formula with no more than k nested adapt operators can be accepted by a counter-free (ω) automaton.*

   For an A-LTL formula with $k+1$ nested adapt operators, it can be expressed as $\phi \stackrel{\Omega}{\rightharpoonup} \psi$, where ϕ and ψ have no more than k nested adapt operators. Based on the assumption, we can build an FSA A for ϕ in finite word domain and a counter-free ω-automata B for ψ in ω-word domain. Then we connect the accepting states of A to the initial states of B with transitions, and make all states in A non-accepting to form an ω-automaton C. Clearly C accepts the set of executions that satisfy $\phi \stackrel{true}{\rightharpoonup} \psi$. By selectively connecting the accepting states of A and the initial state of B, we can make sure all the state pairs satisfy Ω. The resulting ω-automaton accepts exactly the set of sequences satisfying $\phi \stackrel{\Omega}{\rightharpoonup} \psi$. The same process applies to A-LTL in finite word domain.

   Next we show C is also counter-free. Since A and B are counter free, if there

is a counter in C, the counter must include states both in A and in B. Since there is no transition from B to A, such a counter cannot exist. Therefore, C is counter-free.

Therefore, any A-LTL formula with no more than $k+1$ nested adapt operators can be accepted by a counter-free (ω) automaton.

2. *Any counter-free automaton can be expressed in LTL, and any LTL formula is also an A-LTL formula. Therefore, any counter-free automaton can be expressed in A-LTL.*

A logic L_1 is *more expressive* than another logic L_2, denoted as $L_1 \geq L_2$, if and only if for any formula ϕ_2 in L_2, there exists a formula ϕ_1 in L_1 such that ϕ_1 accepts exactly the same set of *models* that ϕ_2 accepts [38]. We say a logic L_1 is *strictly more expressive* than another logic L_2, denoted as $L_1 > L_2$, if and only if $L_1 \geq L_2$ but $L_2 \ngeq L_1$. We say L_1 and L_2 are equivalent in expressive power, denoted $L_1 \equiv L_2$, if and only if $L_1 \geq L_2$ and $L_2 \geq L_1$.

Theorem 2: *A-LTL is equivalent to LTL in expressive power: A-LTL $\equiv$ LTL.*

Proof 2: *We prove that A-LTL is more expressive than LTL and vice versa.*

1. *A-LTL is more expressive than LTL: A-LTL $\geq$ LTL.*

   The proof of this statement is straightforward. Since A-LTL is a super set of LTL, for any formula ϕ of LTL, there is also a formula ϕ' of exactly the same form in A-LTL, and $\phi' = \phi$. Therefore, we have A-LTL $\geq$ LTL.

2. *LTL is more expressive than A-LTL: LTL $\geq$ A-LTL.*

   For any formula ϕ in A-LTL, we can build a counter-free automaton that accepts exactly the set of words S that satisfy ϕ (Theorem 1). And thus, there exists an

LTL formula ϕ' that accepts S. Then we have LTL $\geq$ A-LTL.

3.3 Adaptation Semantics

In this section, we introduce three adaptation semantics specified in terms of A-LTL. In an adaptive program, we term the properties that must be satisfied by the program in each individual domain as the *local properties* for the domain. The properties that must be satisfied by the program throughout its execution, regardless of the adaptations, are called *adaptation global invariants* (or *global invariants*). We use the term *adaptation variants* to denote the properties that change during the program's execution, i.e., from the local properties of the source program to the local properties of the target program. This section focuses on specifying adaptation variants using A-LTL.

Based on results presented in the literature [5, 22, 77] and our own experience [152], we summarize three commonly-used semantics for adaptation. We assume that the local properties of the source program and the target program have both been specified in LTL. We call these local properties *base specifications*. We specify the adaptation from the source program to the target program with A-LTL by extending the base specifications of the source and the target programs. For some adaptations, the source/target program behavior may need to be constrained during the adaptation. These constraints, termed *restriction conditions*, are specified in LTL.

We assume the adaptive program has moderate *computational reflection* capability [90], i.e., it is aware of its adaptation and the currently running steady-state program. This capability can be achieved by simply introducing *flag propositions* in the program to identify its current steady-state program or adaptation status. We assume that a decision maker (as described in Section 2.3) that translates environment

changes into specific adaptation requests is available. Our specification technique describes the expected program behavior in response to these requests. We use an atomic proposition A_{REQ} to represent the receipt of an adaptation request to a target program from the decision maker.

In the following, we summarize three commonly occurring basic adaptation semantic interpretations from the literature [5, 22, 77, 152] specified in terms of A-LTL. There are potentially many other adaptation semantics. In all three adaptation semantics, we denote the source and the target program base specifications as S_{SPEC} and T_{SPEC}, respectively. If applicable, the restriction condition during adaptation is R_{COND}. We assume that the flag propositions to be parts of the specifications. We use the term *fullfilment states* to refer to the states where all the obligations of the source program are fulfilled, thus making it safe to terminate the source behavior.

3.3.1 One-Point Adaptation

Under one-point adaptation semantics, after receiving an adaptation request A_{REQ}, the program adapts to the target program T_{SPEC} at a certain point during its execution. The prerequisite for one-point adaptation is that the source program S_{SPEC} should always eventually reach a fullfilment state during its execution.

$$(S_{SPEC} \wedge \Diamond A_{REQ}) \xrightarrow{\Omega} T_{SPEC}. \tag{3.1}$$

The formula states that the program initially satisfies S_{SPEC}. After receiving an adaptation request, A_{REQ}, it waits until the program reaches a fullfilment state, i.e., all obligations generated by S_{SPEC} are satisfied. Then the program stops being obligated to satisfy S_{SPEC} and starts to satisfy T_{SPEC}. This semantics is visually presented in Figure 3.1(a), where circles represent a sequence of states; solid lines represent state intervals; the label of each solid line represents the property that is held by the

interval; arrows point to the states in which adaptation requests are received. This semantics is straightforward and is explicitly or implicitly applied by most approaches (e.g., [5, 22, 152]) to deal with simple cases that do not require restraining the source behavior or overlapping the source and the target behavior.

3.3.2 Guided Adaptation

Under guided adaptation semantics (visually depicted in Figure 3.1(b)), after receiving an adaptation request, the program first restrains its source program behavior by a restriction condition, R_{COND}, and then adapts to the target program when it reaches a fullfilment state. This semantics is suitable for adaptations whose source programs do not guarantee reaching a fullfilment state within a given amount of time. The restriction condition should ensure that the source program will finally reach a fullfilment state.

$$(S_{SPEC} \wedge (\Diamond A_{REQ} \xrightarrow{\Omega_1} R_{COND})) \xrightarrow{\Omega_2} T_{SPEC}. \tag{3.2}$$

This formula states that initially S_{SPEC} is satisfied. After an adaptation request, A_{REQ}, is received, the program should satisfy a restriction condition R_{COND} (marked with $\xrightarrow{\Omega_1}$). When the program reaches a fullfilment state of the source, the program stops being constrained by S_{SPEC}, and starts to satisfy T_{SPEC} (marked with $\xrightarrow{\Omega_2}$). The *hot-swapping* technique introduced by Appavoo *et al* [5] and the safe adaptation protocol [152] introduced in Chapter 9 use the guided adaptation semantics.

3.3.3 Overlap Adaptation

Under overlap adaptation semantics (visually depicted in Figure 3.1(c)), the target program behavior starts before the source program behavior stops. During the

overlap of the source and the target behavior, a restriction condition is applied to safeguard the correct behavior of the program. This adaptation semantics is appropriate for the case when continuous service from the adaptive program is required. The restriction condition should ensure that the source program reaches a fullfilment state.

$$\Big((S_{SPEC} \wedge (\Diamond A_{REQ} \overset{\Omega_1}{\rightarrow} R_{COND})) \overset{\Omega_2}{\rightarrow} true\Big) \wedge \Big(\Diamond A_{REQ} \overset{\Omega_1}{\rightarrow} (T_{SPEC} \wedge (R_{COND} \overset{\Omega_2}{\rightarrow} true))\Big). \tag{3.3}$$

This formula states that initially S_{SPEC} is satisfied. After an adaptation request, A_{REQ}, is received, the program should start to satisfy T_{SPEC} and also satisfy a restriction condition, R_{COND} (marked with $\overset{\Omega_1}{\rightarrow}$). When the program reaches a fullfilment state of the source program, the program stops being obliged by S_{SPEC} and R_{COND} (marked with $\overset{\Omega_2}{\rightarrow}$). The *graceful adaptation protocol* introduced by Chen *et al* [22] and the *distributed reset protocol* introduced by Kulkarni *et al* [77] use the overlap adaptation semantics.

3.3.4 Safety and Liveness Properties

Temporal logics are often applied to the specifications of *safety* and *liveness* properties of a program. A safety property asserts something bad never happens, while a liveness property asserts something good will eventually happen [124]. Although general forms of safety and liveness properties are not preserved by the adaptation semantics defined above, some common forms of safety and liveness properties are preserved.

We define a formula ϵ to be a *point-safety* property if and only if $\epsilon = \Box\neg\eta$ (read as η never holds during execution), where η is a *point formula* (i.e., a propositional

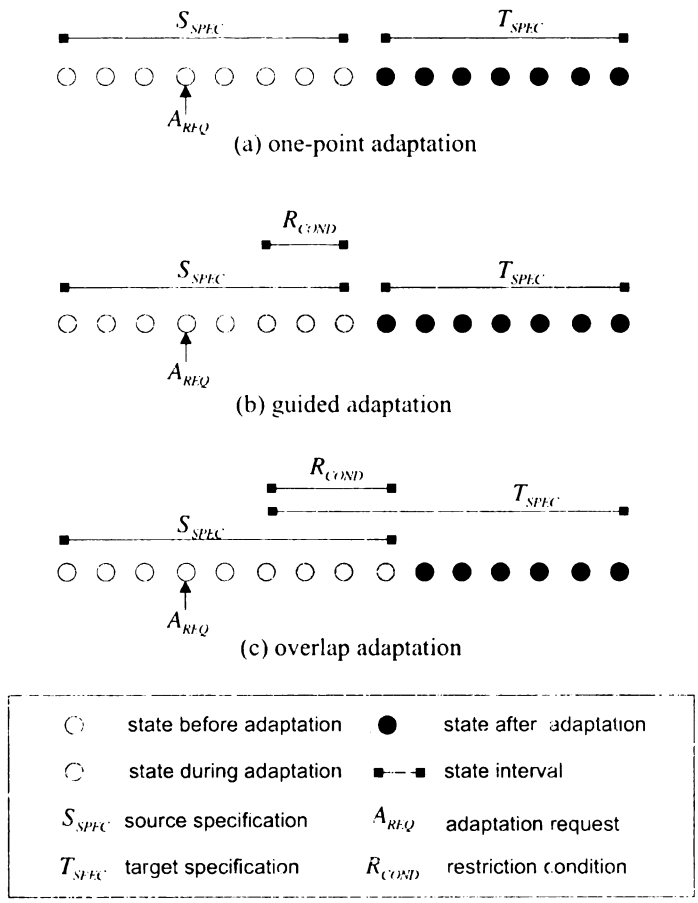


Figure 3.1: Three adaptation semantics

formula). We define a formula ϵ to be *point-liveness* property if and only if $\epsilon = \Box(\alpha \Rightarrow \Diamond\beta)$ (read as it is always the case that if α holds at some point P, then β will eventually hold at a point after P), where both α and β are point formulae.

Theorem 3: *All three adaptation semantics preserve* point-safety *properties. That is, if* $(S_{SPEC} \vee T_{SPEC}) \rightarrow \Box\neg\eta$, *where* η *is a point property, then* $\xi \rightarrow \Box\neg\eta$, *where* ξ *is the adaptation specification based on any one of the three semantics.*

For brevity, we only provide the proof for the one-point adaptation case; other cases can be proved similarly.

Proof 3:

Let the adaptation specification ξ be as in Formula 3.1, that is

$$\xi = (S_{SPEC} \wedge \Diamond A_{REQ}) \stackrel{\Omega}{\rightarrow} T_{SPEC}.$$

For an arbitrary sequence $\sigma \models \xi$, $\exists \sigma'$ and σ'' such that $\sigma' \models S_{SPEC} \wedge \Diamond A_{REQ}$, $\sigma'' \models T_{SPEC}$, and $\sigma = \sigma' \frown \sigma''$. Since $(S_{SPEC} \vee T_{SPEC}) \rightarrow \Box \neg \eta$, we have $\sigma' \models \Box \neg \eta$, and $\sigma'' \models \Box \neg \eta$. Therefore, $\sigma \models \Box \neg \eta$.

This theorem implies that if a propositional property (such as a variable is never greater than a given value) should be held in both the source and the target programs, then the invariant is also held by a simple adaptive program under all three adaptation semantics. This conclusion does not apply to general temporal properties.

Theorem 4: *Point-liveness properties are preserved by all three adaptation semantics. That is, if $(S_{SPEC} \vee T_{SPEC}) \rightarrow \Box(\alpha \rightarrow \Diamond \beta)$, then $\xi \rightarrow \Box(\alpha \rightarrow \Diamond \beta)$, where ξ is the adaptation specification based on any one of the three semantics.*

Again, we only provide the proof for the one-point adaptation case; other cases can be proved similarly.

Proof 4: *Let the adaptation specification ξ be as in Formula 3.1, that is*

$$\xi = (S_{SPEC} \wedge \Diamond A_{REQ}) \stackrel{\Omega}{\rightarrow} T_{SPEC}.$$

For an arbitrary sequence $s_0, s_1, \cdots \models \xi$, $\exists i$, such that $s_0, s_1, \cdots s_i \models S_{SPEC} \wedge \Diamond A_{REQ}$ and $s_{i+1}, s_{i+2}, \cdots \models T_{SPEC}$. Since $(S_{SPEC} \vee T_{SPEC}) \rightarrow \Box(\alpha \rightarrow \Diamond \beta)$, we have $s_0, s_1, \cdots s_i \models \Box(\alpha \rightarrow \Diamond \beta)$, and $s_{i+1}, s_{i+2}, \cdots \models \Box(\alpha \rightarrow \Diamond \beta)$. For an arbitrary state s_j, if $s_j \models \alpha$, then we

have

- *if* $j \leq i$, *then there exists* $k(j < k \leq i)$ *such that* $s_k \models \beta$;
- *if* $j > i$, *then there exists* $k(j < k)$ *such that* $s_k \models \beta$.

That is, $s_0, s_1, \cdots \models \Box(\alpha \rightarrow \Diamond\beta)$.

Therefore, we have $\xi \rightarrow \Box(\alpha \rightarrow \Diamond\beta)$.

3.4 Specification Compositions

Thus far, we have described how to specify simple adaptive programs. These specifications can be composed to describe complex adaptation requirements.

3.4.1 Neighborhood Composition

A complex adaptive program may adapt to different target programs in response to different adaptation requests. The *neighborhood composition* is defined to specify multiple adaptation options from a single program. We define the *neighborhood adaptive program* of a program S to be the union of all the simple adaptive programs that share the same source program S. An execution starting from S can either adapt to a target program if a corresponding adaptation request is received, or remain in S if no adaptation request is received. Assume that the property for the adaptation from S to the i^{th} target program T_i is specified with an A-LTL formula STi_{SPEC}, and the property of S is specified with S_{SPEC}. We can construct the specification for the neighborhood of S by the disjunction of S_{SPEC} and STi_{SPEC}. Let N_{SPEC} be the neighborhood specification of S, we have

$$N_{SPEC} = \bigvee_{i=1}^{k} STi_{SPEC} \vee S_{SPEC}. \tag{3.4}$$

where k is the number of simple adaptive programs sharing the same source program S.

3.4.2 Sequential Composition

A complex adaptive program may sequentially perform adaptations more than once during a single execution. For example, an execution of an adaptive program may start from a program A, then sequentially perform adaptations from A to B to obtain program B, and B to C to obtain program C. This execution should sequentially satisfy the local properties of A, B, and C, and should satisfy the adaptation variants from A to B and from B to C during adaptation. We term the properties that must be satisfied by executions going through multiple steady-state programs *transitional properties*. Specifications for transitional properties can be constructed from adaptation semantics using *sequential compositions* as follows:

Assume that the local properties of A, B, and C are specified with A_{SPEC}, B_{SPEC}, and C_{SPEC}, respectively. The specification for the adaptation from A to B under a given adaptation semantics is a function of A_{SPEC} and B_{SPEC}: $AB_{SPEC} = ADAPT1(A_{SPEC}, B_{SPEC})$. The B to C adaptation specification under a given adaptation semantics is a function of B_{SPEC} and C_{SPEC}: $BC_{SPEC} = ADAPT2(B_{SPEC}, C_{SPEC})$. The specification of A to B to C may be constructed by substituting AB_{SPEC} for B_{SPEC} in BC_{SPEC}.

$$ABC_{SPEC} = ADAPT2(ADAPT1(A_{SPEC}, B_{SPEC}), C_{SPEC}). \tag{3.5}$$

Note that each different adaptation sequence corresponds to a different transitional property. Since in a general adaptive program, there are infinite numbers of different possible adaptation sequences, the number of possible transitional properties is also infinite.

Theorem 5: *Both point-safety and point-liveness properties are preserved by the adaptation semantics and the two types of compositions (i.e., the neighborhood and sequential compositions).*

Proof 5: *Proof outline:*

This theorem can be proved by applying Theorem 3 and Theorem 4.

- *For neighborhood compositions, from Theorems 3 and 4, we know that if all base specifications imply a point-safety (liveness) property ϕ, then all the disjuncts imply ϕ as well. Then the disjunction (the neighborhood composition) also implies ϕ.*

- *For sequential composition, we can prove the conclusion inductively.*

  1. *The base case states that 0-step sequential composition preserves point-safety and point-liveness properties. This has been proved by Theorems 3 and 4.*

  2. *We assume that any n-step sequential composition preserves point safety and liveness properties.*

  3. *An n+1-step sequential composition can be considered as an n-step sequential composition composed with a base specification. Then we can apply the assumption to the n-step adaptation composition case and claim that all n+1-step sequential compositions also preserve point safety and liveness properties.*

3.5 Case Study: MetaSockets

In this section, we use *MetaSockets* [120] as an illustrative example to demonstrate our adaptation specification approach. MetaSockets are constructed from the regular Java Socket and MulticastSocket classes; however, their internal structure and behavior can be modified at run time in response to external conditions. MetaSocket behavior can be adapted through the insertion and removal of *filters* that manipulate the data stream. For example, filters can perform encryption, decryption, forward error correction, compression, and so forth.

We consider a sender's MetaSocket with three different filters: a data compression filter (COM), a DES 64-bit encryption filter (DES64), and a DES 128-bit encryption filter (DES128). The available adaptations are data compression filter insertion and removal, and DES filter replacement. Note that to enforce security, the DES filters can only be replaced by other DES filters, but not be removed. These filters can be combined in four different configurations: DES64, DES128, DES64 with COM (DES64COM), and DES128 with COM (DES128COM). We consider the MetaSocket under each configuration a steady-state program. The adaptive program is initially running in the DES64 configuration.

3.5.1 Specifying Steady-State Programs

The specifications of the programs are described as follows:

- DES64 program:

$$DES64_{SPEC} = (\Box\, DES64_{FL}) \wedge \Box(DES64Input(x) \rightarrow \Diamond DES64Output(x)).$$

The $DES64Input$ (resp. $DES64Output$) are events indicating the input (resp.

output) of a packet to (resp. from) the MetaSocket under DES64 configuration.[1] The flag proposition $DES64_{FL}$ indicates that the program is running under the DES64 configuration. The formula states that under this configuration, for every input packet to be encoded by the DES64 filter, the MetaSocket should eventually output a DES64 encoded packet. The following program specifications can be interpreted in a similar way.

- DES128 program:

$$DES128_{SPEC} = (\Box\, DES128_{FL}) \wedge (\Box(DES128Input(x) \rightarrow \Diamond DES128Output(x))).$$

- DES64COM program:

$$DES64COM_{SPEC} =$$
$$(\Box\, DES64COM_{FL}) \wedge (\Box(DES64COMInput(x) \rightarrow \Diamond DES64COMOutput(x))).$$

- DES128COM program:

$$DES128COM_{SPEC} =$$
$$(\Box\, DES128COM_{FL})$$
$$\wedge(\Box(DES128COMInput(x) \rightarrow \Diamond DES128COMOutput(x))).$$

3.5.2 Specifying Adaptations

To determine the semantics of each adaptation, we consider the following three factors: (1) The MetaSocket component is designed for the transmission of live video and audio data. Therefore, we should minimize data blocking. (2) We should not

[1]Strictly speaking, the notation of DES64Input(x) is a predicate. However, LTL requires the underlying logic to be propositional. Here we implicitly employ the *2-order data abstraction* introduced by Dwyer and Pasareanu [36], which converts predicates to propositions by using constant values to represent arbitrary values.

allow both the source and the target programs to input simultaneously because that will cause ambiguity in the input. (3) We should not allow the target program to output data before any output is produced from the source program, otherwise, it will complicate the logic on the receiver. Based on the above considerations, it is appropriate to apply the overlap semantics with conditions prohibiting the types of overlap discussed above.

We use the DES64 to DES128 adaptation as an example to demonstrate the simple adaptive program specification construction. The restriction condition is that neither the input nor the output should overlap between the source and the target programs. Formally in LTL:

$$R_{COND}(DES64\text{-}DES128) = \Box(\neg DES64Input(x) \wedge \neg DES128Output(x)). \qquad (3.6)$$

The intuition for this restriction condition is that the program should not accept any more DES64 inputs and will not produce any DES128 outputs until all DES64 outputs have been produced. Given the source and target program specifications, the overlap semantics, and the restriction condition, we apply Formula 3.3 to derive the following specification:

$$\begin{aligned} &DES64\text{-}DES128_{SPEC} = \\ &\Big(\Big(DES64_{SPEC} \wedge (\Diamond A_{REQ}(DES128) \rightarrow R_{COND}(DES64\text{-}DES128))\Big) \dashrightarrow true\Big) \\ &\wedge \\ &\Big(A_{REQ}(DES128) \rightarrow (DES128_{SPEC} \wedge (R_{COND}(DES64\text{-}DES128) \rightarrow true))\Big). \end{aligned} \qquad (3.7)$$

Formula (3.7) states that after the program receives an adaptation request to the DES128 program ($A_{REQ}(DES128)$), it should adapt to the DES128 program. Further-

more, the behavior of DES64 and DES128 may overlap, and during the overlapping period, the program should satisfy the restriction condition $R_{COND}(DES64\text{-}DES128)$. In this example, the Ω notation of the adaptation operators is not used. We simply assign *true* to Ω in the four adapt operator locations. With the same approach, we specify other simple adaptive programs in the adaptive program. Further more, both the source and the target specifications are point liveness properties. We can unify them with the following formula by disregarding the types of inputs and outputs.

$$\Box(Input(x) \rightarrow \Diamond Output(x)). \tag{3.8}$$

According to Theorem 4, we can conclude that the adaptation also satisfies the point liveness property.

3.5.3 Specifying Global Invariants

In addition to the specification for adaptation behavior, the program should also satisfy a set of adaptation invariants to maintain its integrity

- **Security invariant**: At any time during execution, insecure output should not be produced, i.e., all output packets should be encoded by a DES filter:

$$INV1 = \Box\neg insecureOutput(x).$$

- **QoS invariant**: During program execution, the sender MetaSocket should not cause any loss of packets, i.e., all input packets should be output:

$$INV2 = \Box(Input(x) \rightarrow \Diamond Output(x)).$$

Note, this invariant is already implied by the point liveness preservation prop-

erty of the adaptation semantics (Formula 3.8).

- **Precedence invariant**: The MetaSocket should not output a packet before the corresponding input has been received:

$$INV3 = \Box(\neg Output(x)\,\mathcal{U}\,Input(x)).$$

3.6 Related Work

We now introduce work related to A-LTL and the adaptation semantics discussed in this chapter, including work related to the classification of adaptation semantics, and work related to the specification of critical properties in adaptive software.

Biyani *et al* [14] classified distributed software adaptation into *overlap adaptation*, where the source and the target programs overlap during adaptation, and *non-overlap adaptation*, where the source and the target programs are not present in the system simultaneously during adaptation. They further refined the overlap adaptation into three sub-categories: *quiescence* adaptation, *parallel* adaptation, and *mixed-mode* adaptation. In both quiescence and parallel adaptations, the source and the target programs are not allowed to communicate with each other. The key difference is that although quiescence adaptation allows the source and the target programs to coexist in the system, they are not allowed to coexist in a single process, while parallel adaptation allows both. In mixed-mode adaptation, the source program is allowed to directly communicate with the target program. Their classification is largely inspired by ours, where their non-overlap adaptation subsumes both the one-point adaptation and the guided adaptation semantics as described in this chapter. The different types of adaptation they introduced can be specified by our approach. For example, in the quiescence adaptation, the exclusive relationship between the source and the target components in each process can be specified by an overlap adaptation

semantics with restriction conditions preventing the different versions of the same functionality from coexisting. The parallel adaptation can be specified similarly with the restriction conditions relaxed to allow different versions of the same functionality to coexist. The mixed-mode adaptation can be modeled as a sequential composition of two one-point adaptations: from S_{SPEC} to M_{SPEC}, and from M_{SPEC} to T_{SPEC}, where the intermediate specification M_{SPEC} allows the source program to communicate with the target program. Their focus is to study assurance techniques for the mixed-mode adaptation, rather than to specify adaptation semantics using a temporal logic as in our approach.

Other temporal logics have also been proposed to specify properties of adaptive software. Feather *et al* [41] proposed using a real-time temporal logic, FLEA [31], to specify consistency constraints in adaptive software. The FLEA specification language describes orders and durations of events using five basic temporal operators: **OR**, **THEN**, **COUNT**, **WITHIN**, and **START**. A FLEA compiler automatically converts a FLEA expression into run-time monitoring code that verifies the conformance between a sequence of events and the FLEA expression. Compared to A-LTL, FLEA has the capability of expressing quantitative timing constraints, as in other real-time temporal logics [3, 48, 107]. However, like LTL, FLEA is not convenient to specify adaptation requirements since it is not designed specifically for that purpose. The properties they introduced in their paper were all non-adaptive properties rather than adaptation properties as described in this chapter.

Other non-temporal logic formal specification languages have also been proposed to specify properties of adaptive software. Kramer *et al* [72] described how to use a process algebraic language, *FSP*, to specify the critical properties that must be satisfied by the adaptive software described in *Darwin*. A critical property is specified as an FSP process, which identifies a set of allowable action sequences. New FSP processes can be composed from existing ones using primitive operators including

prefix ("→"), choice ("*mid*"), and parallel ("||"). They demonstrated using FSP to specify that a propositional property is satisfied in certain states. Allen *et al* [2] used *Dynamic Wright* to specify the behavior of adaptive software, and used a process algebraic language, *CSP* [58], to describe the critical properties that must be satisfied by the software. Since process algebras are generally considered as design-level languages, the properties that the above two approaches are concerned about are also design-level properties. In contrast, the A-LTL introduced in this chapter is intended for expressing requirements-level properties. Furthermore, the properties they specified are not used to describe how the system may change, i.e., adaptation variants. Instead, they are concerned about global invariants. In contrast, we specify both global invariants and adaptation variants in adaptive software with A-LTL/LTL.

3.7 Discussion

Adaptation semantics must be precisely specified at the requirements level so that they can be well understood and correctly implemented in later phases of the software development process. After an adaptation temporal specification is constructed, it can serve as guidance for the adaptation developers to clarify the intent for the adaptive program. It also enables us to perform model checking to verify the correctness of the program model against the temporal logic specifications with both static verification and run-time model checking.

Our technique can be used to specify adaptation behavior of autonomic computing systems to achieve self-healing, self-protecting, self-managing, and so on [24, 33, 50]. Although environment changes such as failures are unexpected, after the decision maker has translated the changes into adaptation requests, the program behavior in response to the requests is generally deterministic. Our technique can be applied to describe the expected program behavior of adaptive programs in response

to these adaptation requests.

We chose to use the A-LTL to specify adaptation semantics due to two major considerations: effectiveness and simplicity. Compared to LTL, A-LTL is more effective in conveying adaptation objectives in specifications. Some other logics, including the *choppy logic* [119], the propositional interval temporal logic (PITL) [17, 52, 102], and the interval temporal logic (ITL) [20], are also sufficient to specify adaptation behavior. However, they are generally more complex and verbose. We find A-LTL to be simple and sufficiently effective to specify adaptation behavior, given that the logic is specifically designed to support adaptation.

A-LTL specifies only the relative temporal ordering among events and system states that occur during the adaptation process. While this capability is suited to specifying many types of adaptive behavior, A-LTL is insufficient to specify systems with real-time requirements, where the absolute timing may play an important role. To address this shortcoming, we developed TA-LTL, a real-time extension to A-LTL; the details of the specification language are introduced in Appendix A.

Chapter 4

Goal-Based Requirements Analysis

In this chapter, we introduce a goal-based approach [32, 84] to construct LTL and A-LTL requirements specifications for adaptive software. In Chapter 3, we have introduced the temporal logic, A-LTL, to specify adaptation semantics from one steady-state behavior to another. In this chapter, we discuss the technique to determine the different steady-state behaviors the adaptive program should be able to exhibit, and the adaptations among these behaviors. As a result, we produce adaptation specifications in A-LTL/LTL for the adaptive program, which can be used in the design phase for formal analysis. This chapter is organized as follows. Section 4.1 gives background information on goal-based models. Section 4.2 describes the goal-base requirements specification for adaptive software. Related work is described in Section 4.3. Finally, Section 4.4 summarizes this chapter.

4.1 Background: Goal-Based Models

In this section, we briefly overview relevant knowledge of the goal-based specification language [32, 84] used in this chapter. A *goal model* specifies *goals* that are stakeholder objectives that the system should achieve. A goal may be refined into subgoals and/or requirements that elaborate how the goal is achieved. A goal

is *AND-refined* if its subgoals must *all* be achieved for the goal itself to be achieved. A goal is *OR-refined* if any one of its subgoals must be achieved for the goal itself to be achieved, i.e., OR-refinement describes alternative ways to achieve the goal. A *requirement* is a goal that can be directly operationalized by an *implementation*. A *condition* for a goal is the property that is assumed to be *true*. Existing goal model representations include KAOS [13] and Tropos/i* [140], either of which is sufficient for our specification purpose.

4.2 Specifying Adaptation Requirements

we now describe a goal-based requirements specification technique that constructs temporal logic specifications for adaptive software. We first describe the process to construct global invariants and local properties in LTL. Then we introduce the use of an adaptation semantics graph to construct adaptation specifications in A-LTL.

4.2.1 Local Properties and Global Invariants

We use a goal-based approach [82, 134] to analyze global invariants and local properties in adaptive software [144]. Berry *et al* [10] have formalized the different environmental conditions in which the program is required to execute as a set of execution *domains* $D_1, D_2, \cdots, D_k$. As shown in Figure 4.1, we create a goal model for the adaptive software, where the top-level goal is intended to be achieved by the adaptive software under different run-time environmental conditions. We start by creating a high-level goal node in a goal model. Then we study the requirements for the adaptive program and refine the goal model according to the following steps.

1. We use high-level specification languages, e.g., LTL, to specify the properties (i.e., global invariants) the adaptive program must satisfy throughout its execu-

tion in order to achieve the top-level goal. The global invariants usually contain safety and liveness constraints.

Model refinement: We create a requirement node *GI* (depicted as a parallelogram) containing the global invariants under the top-level goal in the goal model.

2. We determine the set of environmental conditions (domains) in which the adaptive program is required to execute (e.g., data loss conditions of a communication channel).

   Model refinement: For each different domain, we create an OR-refined subgoal node (e.g., *subgoal j*) under the top-level goal. Under each subgoal node, we create a condition node (e.g., D_j) to include the conditions for the domain.

3. We use high-level specification languages, e.g., LTL, to specify the local properties in each domain. The local properties must be satisfied by the adaptive program in order to achieve the top-level goals under the conditions of the domain.

   Model refinement: We create a requirement node R_i containing the local properties under each subgoal in the goal model.

4.2.2 Adaptation Variants

Next, we construct A-LTL specifications for adaptation variants. We study 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 (where P_i satisfies local property R_i). A change of domain from D_i to D_j may warrant an adaptation of the program from P_i to P_j (where P_j satisfies R_j) depending on the cost to develop such an adaptation and the overhead that may be

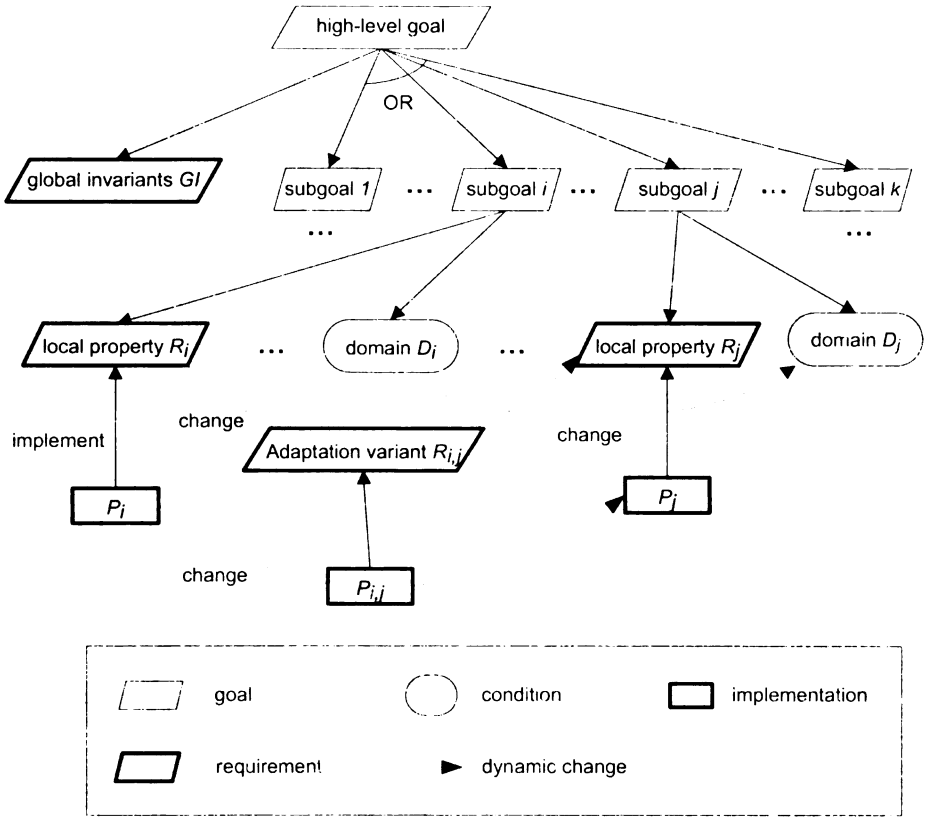


Figure 4.1: Goal model for adaptive software

incurred during the adaptation. For each adaptation that moves from satisfying R_i to satisfying R_j, we determine its adaptation semantics based on the characteristics of the adaptive program, e.g., whether certain functionalities can be temporarily disabled during adaptation, etc.

After the adaptation semantics of all adaptations are determined, the adaptation variants of the adaptive program can be visually represented as a graph, where each different local property is depicted by a vertex and each adaptation is depicted by an arc. For example, the adaptation semantics graph for the MetaSockets introduced in Section 3.5 is visually represented in Figure 4.2. This graph represents that the program is required to adapt among the different local properties using the overlap

adaptation semantics.

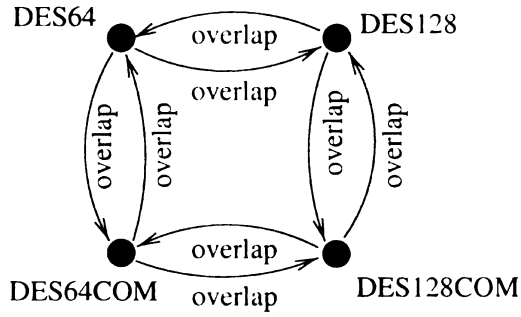


Figure 4.2: Adaptation semantics graph

Formally, we define an *adaptation semantics graph* to be a tuple $(\Phi, \Phi_0, A, \Psi, INV)$, where

- Φ is a set of vertices representing the set of local properties for an adaptive program.
- Φ_0 ($\Phi_0 \subseteq \Phi$) is a set of initial vertices, representing the initial local properties of the adaptive program.
- A ($A \subseteq \Phi \times \Phi$) is a set of arcs representing adaptations.
- Ψ ($\Psi : A \rightarrow$ semantics) is a function that maps an adaptation to the semantics for the adaptation.
- INV is the set of adaptation invariants.

We represent an adaptation semantics with an A-LTL formula template where symbols are placeholders that are replaced with specifications when the semantics is instantiated for a specific adaptation. For example, the one-point adaptation semantics is represented as

$$(S_{SPEC} \wedge \Diamond A_{REQ}) \xrightarrow{\Omega} T_{SPEC},$$

where S_{SPEC}, A_{REQ}, Ω, and T_{SPEC} are symbols. When this semantics is evaluated upon the following symbol assignments:

$$\begin{aligned} S_{SPEC} &\equiv DES64_{SPEC}, \\ T_{SPEC} &\equiv DES128_{SPEC}, \\ A_{REQ} &\equiv REQ_{DES64-128}, \\ \Omega &\equiv true. \end{aligned}$$

We derive the DES64 to DES128 one-point adaptation specification by replacing the symbols in the expression with their corresponding values:

$$(DES64_{SPEC} \wedge \Diamond REQ_{DES64-128}) \overset{true}{\rightsquigarrow} DES128_{SPEC}.$$

Adaptation specifications for the adaptive program in A-LTL can be evaluated from the adaptation semantics graph using the simple symbol substitution described above.

4.3 Related Work

The goal-based approach introduced in this chapter is influenced by many others [41, 82, 142] who have applied goal-based models to analyze adaptive software. Lapouchnian *et al* [82] introduced the use of goal models as a foundation for adaptive software development process, and as a possible architecture sketch for autonomic systems that can be built from these models. They use goal models to explicitly represent and analyze the space of alternative ways to fulfill a stakeholder goal. They distinguish two types of goals: *Hard goals* are functional stakeholder goals that describe the tasks to be accomplished (e.g., schedule a meeting); *softgoals* are qualitative attributes that describe the quality of the tasks (e.g., good quality schedule). The

goal models are used to guide the design and implementation of adaptive programs. Feather *et al* [41] proposed using a goal-based approach to model and monitor the run-time behavior of adaptive software. The KAOS goal modeling language they used has a two-level structure: The *outer semantics net* layer is the graphic goal representations used for declaring concepts, their attributes, and various links among the concepts. An *internal formal assertion* layer formally defines the concepts using temporal logics. The critical properties in a goal model are specified using a real-time temporal logic [70] in the internal formal assertion layer. The temporal logic specifications are then translated into FLEA [31] specifications, which can be converted into run-time monitoring code by the FLEA compiler [31] for run-time verification. The properties they specified were all global invariants, i.e., non-adaptive properties. Our approach is inspired by the approaches described above. We extended their approaches with the use of A-LTL for specifying global invariants, local properties, and adaptation variants properties. We assigned each potential adaptation an adaptation semantics and proposed the adaptation semantics graph to represent adaptation variants and automatically generate adaptation specifications for selected adaptations.

4.4 Discussion

In this chapter, we introduced the use of goal-based models and adaptation semantics graphs to analyze and specify the adaptation semantics of adaptive programs in A-LTL/LTL. These techniques enable adaptation developers to achieve the benefits of formal specification while using the easier-to-understand graphical notations.

Our goal-based adaptive software requirements analysis approach is inspired by existing goal-based analysis techniques, including KAOS [41] and Tropos/i* [141]. Our approach extends these techniques by associating the goal-based models with A-LTL specifications.

The goal-based requirements analysis approach introduced in this chapter serves as the foundation of later chapters in this thesis. A number of concrete examples of its applications will be shown in Chapters 5 and 6.

Part II

Model Design and Analysis of Adaptive Software

Chapter 5

MASD: Model-Based Adaptive Software Development

In this chapter, we propose a process for designing formal adaptation models for adaptive software and ensuring that these models satisfy the adaptation requirements constructed using the specification approach introduced in Part I (Chapters 2–4). We first apply the goal-based technique to specify the adaptation requirements for adaptive software. We then construct formal adaptation models for adaptation, and verify these models against the above requirements. These formal adaptation models are used as blueprints for the implementation of adaptive software.

Numerous research efforts have been proposed to formally specify dynamically adaptive programs in the past several years [18]. Graph-based approaches model the dynamic architectures of adaptive programs as graph transformations [57, 99, 128]. Architecture Description Language (ADL)-based approaches model adaptive programs with connection and reconnection of connectors [73, 106, 131]. 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 [2, 19, 72]. 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, therefore, the target behavior after adaptation must always start from the initial state, thus making adaptations less flexible; (3) These techniques are specific to systems specified by a special type of formalism (i.e., process algebra), making them potentially difficult to be extended to systems specified by other types of formalisms.

To address the above drawbacks in existing approaches, we introduce a Model-based Adaptive Software Development (MASD) approach. Our approach has the following features that distinguish it from the existing approaches. (1) Our approach separately models the non-adaptive behavior and the adaptive behavior, thus making each of the respective models simple and more amenable to automated analysis and visual inspection; (2) We use global invariants to specify properties that should be satisfied by adaptive programs regardless of adaptations. These properties 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) and UML state diagrams; (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 implementation.

This chapter 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 terms of the program behavior before, during, and after adaptation, the requirements for the adaptive program, and the adaptation mechanism. This kind of definition leads to precise and flexible adaptation specifications.

We accompany the discussion of our approach with an example of an adaptive GSM (Global System for Mobile Communications)-oriented audio streaming protocol [154].[1] This protocol had been previously developed without our technique, where the developers had found the overall program logic to be complex and error-prone. Our approach significantly reduced the developer's burden by separating different concerns and leveraging automated model construction and analysis tools, and thus decreased the development time and improved software quality.

The remainder of this chapter is organized as follows. Section 5.1 gives background information on Petri nets, the modeling language used in this chapter. Section 5.2 describes our approach for constructing and analyzing models for adaptive programs. Section 5.3 outlines the generation of adaptive programs based on the models. Section 5.4 describes an adaptive Java pipeline case study using the proposed model-based development technique. Related work is introduced in Section 5.5 and Section 5.6 discusses limitations and possible extensions of the proposed approach.

5.1 Background: Petri Nets

This section briefly overviews Petri nets which we use to model adaptive systems. *Petri nets* are a graphical formal modeling language [109, 110], where a Petri net comprises *places*, *transitions*, and *arcs*. Places may contain zero or more tokens. A state (i.e., a *marking*) of a Petri net is given by a function that assigns each place the

[1] GSM is a telephone audio compression standard defined by the European Telecommunications Standards Institute [40].

number of tokens in the 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, where firing a transition has the following three effects: (1) consuming the tokens from its input places, (2) performing the associated actions, and (3) 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. For example, the extension used in this chapter, *Coloured Petri Nets* (CPN) [75] allow the tokens to be distinguished by their colors (or types), making them 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 input (or output) 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* [94] is a coloured Petri nets analysis tool that supports model checking for LTL properties. *Renew* [78] 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 enables Petri net models to communicate with each other and with other Java applications. Figure 5.1 depicts a coloured Petri net created with Renew, where circles P1 and P2 represent places, and the box T represents a transition. P1 and P2 are the input and output places of T, respectively. The inscription of P1 ("[2]") indicates that P1 contains an integer type token with value 2.

The inscriptions of the input and the output arcs for T ("x" and "x+1", respectively) show the relationship between the input and the output of T. The inscription of T ("guard $x < 3$") shows the condition under which the transition is enabled. In this example, T is enabled, and firing the transition consumes the token in P1 and generates an integer token in P2 with value 3. In this chapter, coloured Petri nets are used to specify the design of adaptive systems.

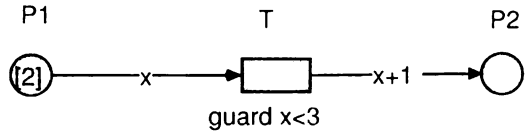


Figure 5.1: A coloured Petri net example.

5.2 Our Specification Approach

We propose a general specification process for adaptive systems using state-based modeling languages. We also describe 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 communication protocol example. Although we illustrate our approach with Petri nets, our approach extends to other state-based modeling languages, including UML Statechart diagrams [149] (Chapter 6).

The approach introduced in this section leverages the artifacts produced in Part I, where we proposed a goal-based requirements analysis to construct adaptation specifications in A-LTL/LTL. In this section, we assume that the requirements analysis has generated the following artifacts (shown in Figure 5.2):

- A set of global invariants *INV*, specified in LTL.

- A set of domains $D_1, D_2, \cdots, D_n$ in which the program is required to execute, and the local requirements for each domain $R_1, R_2, \cdots, R_n$, specified in LTL.
- A set of adaptations $A_{i,j}$ that the program is required to perform, and the adaptation variant property for each adaptation $R_{i,j}$.

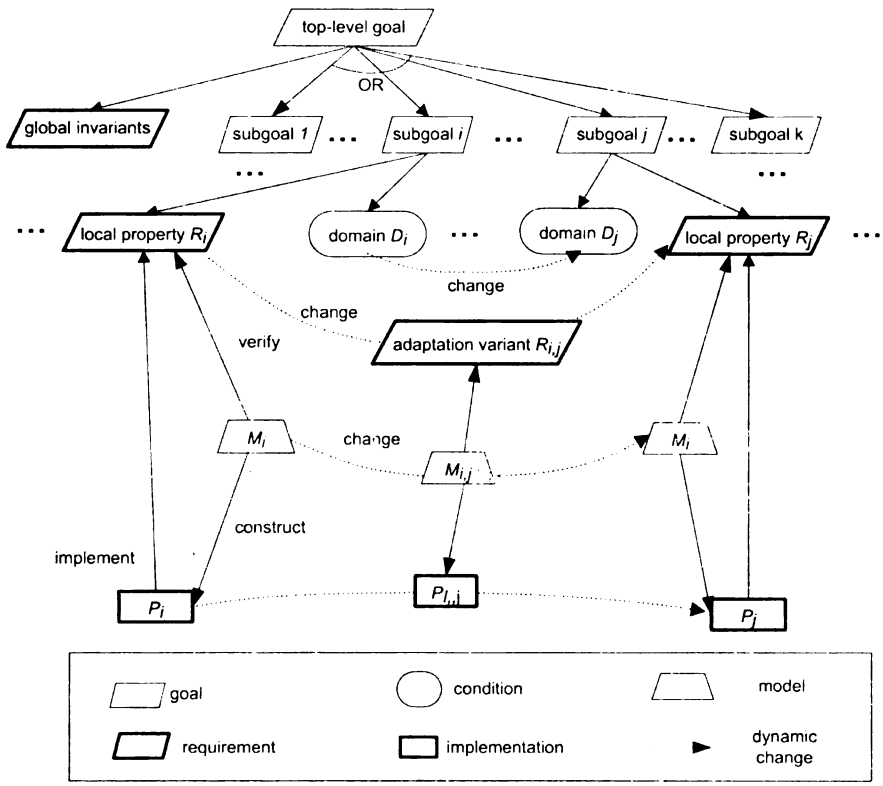


Figure 5.2: Goal model for adaptive software

In this chapter, we focus on constructing adaptation design models that satisfy the above requirements (also shown in Figure 5.2). This is achieved in the following steps :

Step 1: For each domain D_i, construct a state-based model M_i. Verify the model against the local properties of the domain R_i using model checking or simulation.

Step 2: For each adaptation $A_{i,j}$, construct an adaptation model $M_{i,j}$ for the adaptation. Verify the adaptation model against global invariants INV and adaptation variants $R_{i,j}$.

Step 3: These state-based adaptation 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.

When errors are found in a given step, developers may be required to return to an earlier step to correct the errors.

We introduce the general process to model and analyze an adaptive program in Petri nets, accompanied by a concrete GSM (Global System for Mobile Communications)-oriented audio streaming protocol.[2]

5.2.1 Illustrative Adaptation Scenario

GSM-oriented audio stream encoding and decoding protocol is a signal processing-based forward error correction protocol [16, 154]. 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 5.3 shows a GSM-oriented audio streaming protocol example with $\theta = 1$, $c = 2$. In order to

[2]In this chapter, we only discuss the verification of local properties and global invariants. The verification of adaptation variant properties is discussed in Chapter 7.

accommodate packet loss changes in the network connection, we dynamically switch among components that implement different θ and c values.

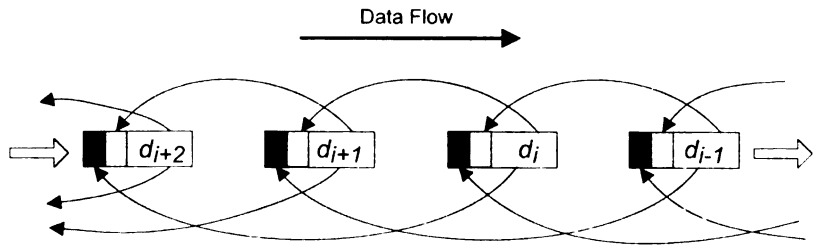


Figure 5.3: GSM-oriented encoding and decoding

The configuration of the system is shown in Figure 5.4. 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, the sender/receiver should use a high loss-tolerance and consequently high bandwidth-consuming protocol. Specifically, in this chapter, 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. Note that this adaptation may appear to be simple parameter changing, where, in fact, it requires swapping encoders and decoder at run time. Moreover, the states of the GSM(1,2) encoder/decoder must be correctly preserved and transferred to the states of the GSM(1,3) encoder/decoder so that the system is always running in a consistent state.

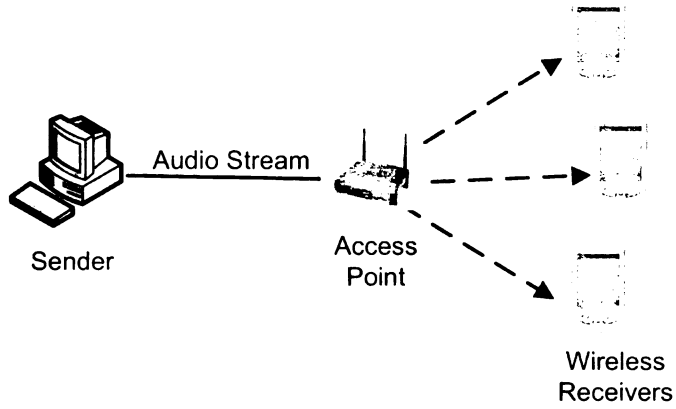


Figure 5.4: Audio streaming system connection

5.2.2 Constructing Models for Source and Target

Assume that we 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). The requirements R_S for the source domain are:

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

$$\Diamond(dataSource = empty) \wedge \Box(read(x) \rightarrow \Diamond send(x)).^{3} \qquad (5.1)$$

[3]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 *propositionalization* [127], which

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

$$\Box(receive(x) \rightarrow \Diamond decode(x)). \tag{5.2}$$

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

$$(\Box lossCount <= 2) \rightarrow (\Box \neg lose(x)). \tag{5.3}$$

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

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

$$(\Box lossCount <= 3) \rightarrow \Box(\neg lose(x)). \tag{5.4}$$

To model the program for S, we build a Petri net for the GSM(1,2) encoder on the sender side and a Petri net for the GSM(1,2) decoder on the receiver side. Figure 5.5 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 either by 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

converts predicates to propositions by using constant values to represent arbitrary values.

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 output socket place.

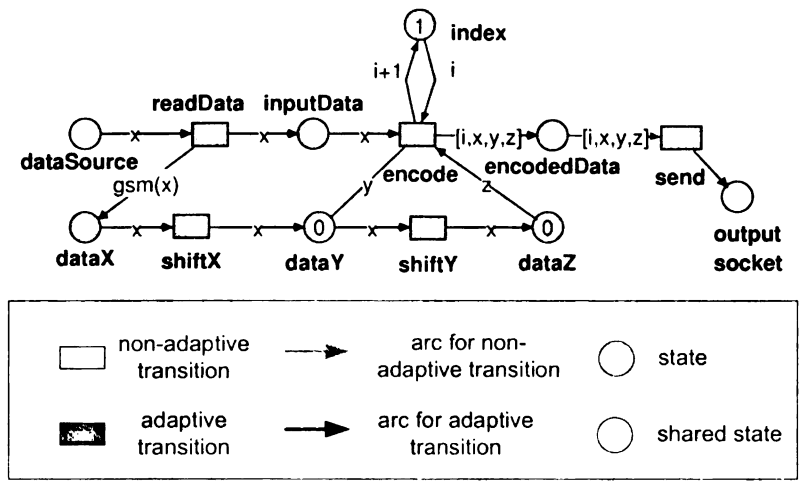


Figure 5.5: Sender source net (GSM (1,2))

Figure 5.6 shows the receiver net (elided). The receive transition receives the audio packets from the input socket 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.

We build the model for the lossy network separately (shown in Figure 5.7); this model serves to define the behavior of the environment in which the sender and the receiver execute. The input socket and output socket places are shared by the sender and receiver nets, respectively. The lossy network is modeled as a packet queue. Firing the send transition puts a data packet in the output socket place, thereby enabling both the enqueue transition and the lose transition. Firing the lose transition discards

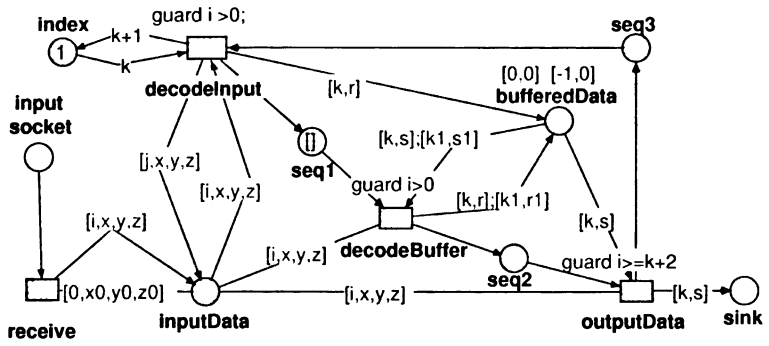


Figure 5.6: Receiver source net (GSM (1,2))

the packet in the output socket place and increments the value in the loss count place. Firing the enqueue transition moves a packet from output socket to the network buffer place. Firing the dequeue transition moves a packet from network buffer to input socket of the receiver. The packet in input socket can be received by the receiver. The value in the loss count place indicates the number of packets lost during data transmission.

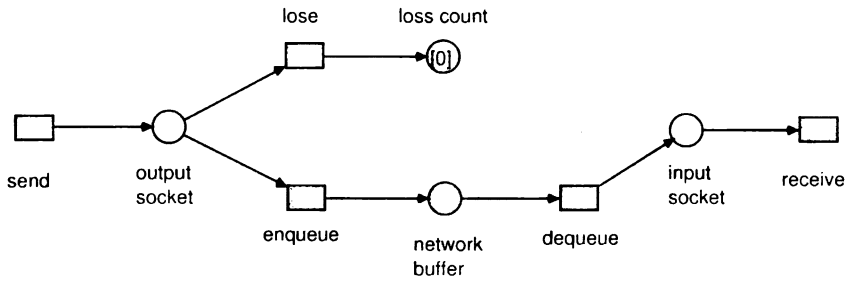


Figure 5.7: Lossy network net (environment model)

After building the source models, we first play token games with the sender and receiver models (prior to adaptation) 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 (Formulae 5.1–5.3). 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 5.8 and 5.9 show the sender net and the receiver net, respectively.[4] The modeling, validation, and verification process is similar to that used for the source model.

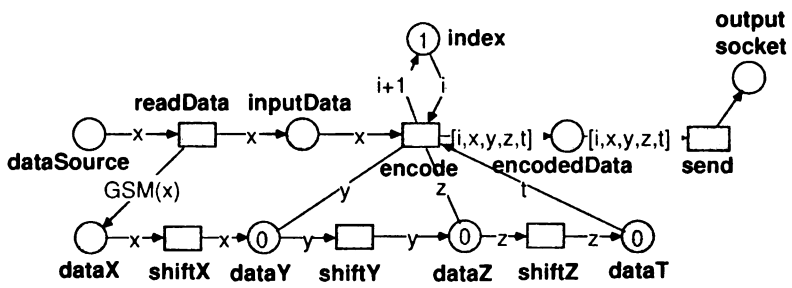


Figure 5.8: Sender target net (GSM (1,3))

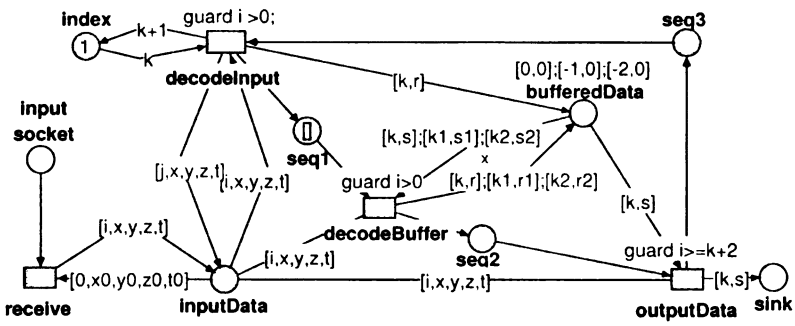


Figure 5.9: Receiver target net (GSM (1,3))

[4]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.

5.2.3 Constructing Adaptation Models

This section describes Step (2), the process to create a model of the adaptation behavior from the source domain to the target domain. Recall the three types of adaptation semantics introduced in Chapter 3: *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 approach for state-based modeling languages, then instantiate the approach with Petri nets and apply it to the GSM-oriented protocol example.

The term *quiescent states* is commonly used to refer to those states suitable for adaptations in the literature [5, 71]. 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 those needed for 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.

We argue that the quiescent states of an adaptive program must be defined in the context of the program adaptation behavior before, during, and after adaptation, and the global invariants that the adaptive program must satisfy. 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 adaptive transition $s \rightarrow t$ 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 the quiescent state identification. 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 in the source net, and all the output places are in 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 in a similar fashion, 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 need to revise the models and/or the properties.

One-point adaptation

As described in Section 3.3, with one-point adaptation, at one state during the source program's execution, the source behavior should complete, and the target behavior should start. 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.

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,

$$\Diamond(dataSource = empty) \wedge \Box(read(x) \rightarrow \Diamond send(x)). \tag{5.5}$$

- **Receiver global invariant:** The receiver should always decode data once a new packet is received. In LTL,

$$\Box(receive(x) \rightarrow \Diamond decode(x)). \tag{5.6}$$

The adaptation model for the sender is shown in Figure 5.10. 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 5.11 shows the adaptation model for the receiver. The quiescent states of the receiver are identified by the adapt transition of the receiver. 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 in order to validate that the models reflect our purpose for the program. Then we used Maria to model check these models against the global invariants (Formulae 5.5 and 5.6). Once the models passed our validation and verification analysis, we concluded that these

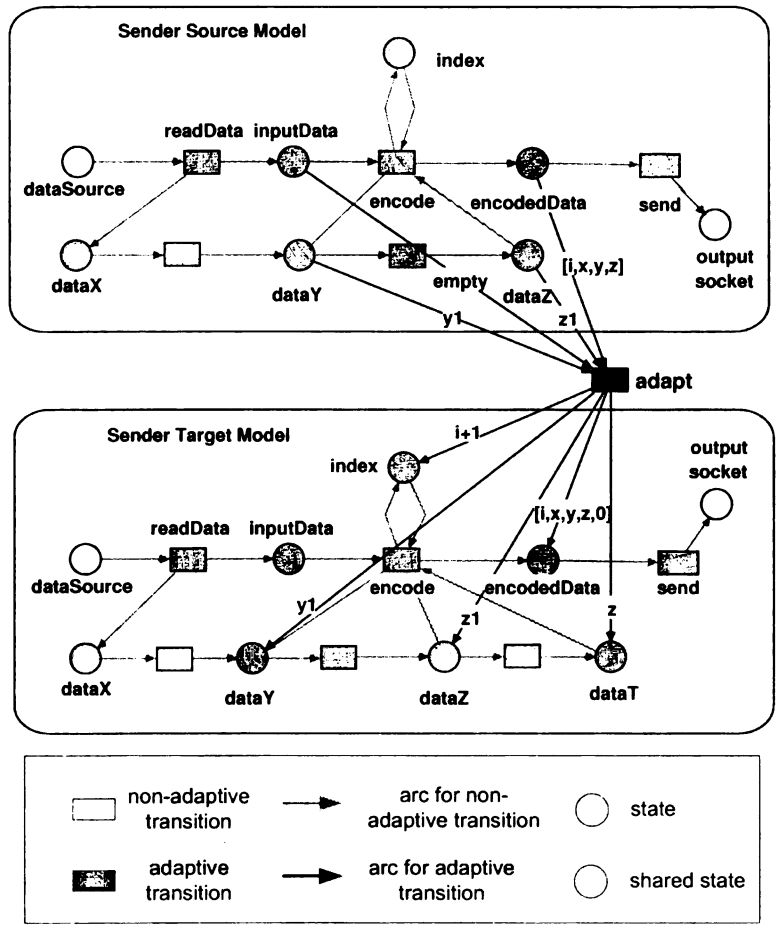


Figure 5.10: Sender adaptation net

models had been constructed appropriately.

Guided adaptation

As described in Section 3.3, with guided adaptation, when the source program receives an adaptation request, it enters a restricted mode, in which some functionalities of the program are blocked. Entering the restricted mode ensures that the program will eventually reach a quiescent state, from which a one-point adaptation takes the program to the target program state space.

To specify a guided adaptation, we should determine the functionalities that

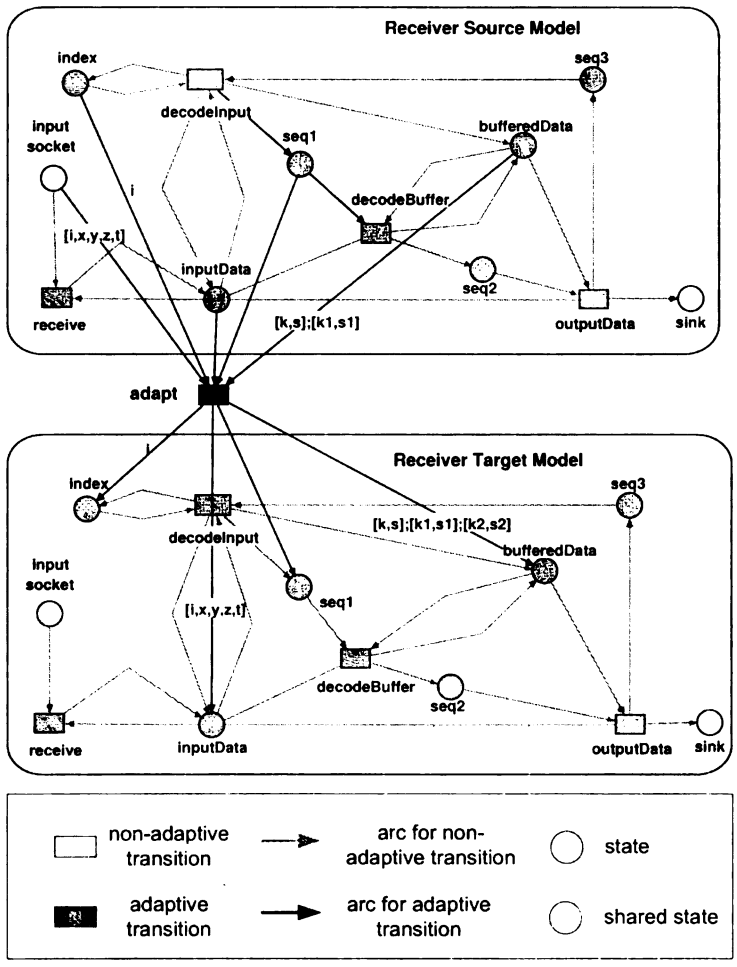


Figure 5.11: Receiver adaptation net

should be blocked in the restricted mode, and identify the quiescent states of the program in the restricted mode. To achieve functionality blocking, transitions are removed 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 approach to define quiescent states of M'_S and the state transformation from M'_S to M_T is the same as that used for one-point adaptation.

The approach in which 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 5.10, which requires the **inputData** place to be empty, the **encodedData** place to be non-empty, and the encoded data in **dataY** and **dataZ** to have already shifted one location. The semantics of Petri nets determines that the order of 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 that represents a restricted variation of the source net that disables the **send** transition. We do this by copying the source net, then removing the **send** transition from the net. Figure 5.12 shows the conceptual model for 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 5.10. Note, the way in which

N is constructed guarantees that all properties verified before are still valid. The only additional property we need to verify is that N will eventually reach a quiescent state, which is formally specified as follows:

- **Quiescence constraint:** After the restrict transition is fired, the restricted sender must eventually reach a quiescent state. In LTL,

$$\Box(restrict \Rightarrow \Diamond quiescent). \tag{5.7}$$

We model checked the model in Figure 5.12 against Formulae 5.5–5.7, where the analysis verified that the property holds.

Overlap adaptation

As described in Section 3.3, 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 before the last adaptive transition. Overlap adaptation is particularly useful for 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 yields overlap adaptation.

Overlap adaptation is more complex than one-point adaptation in the sense that we need to determine not only the state transformation functions of each one-point or guided adaptation, 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 specified global invariants, an adaptive program should also satisfy an *adaptation integrity*

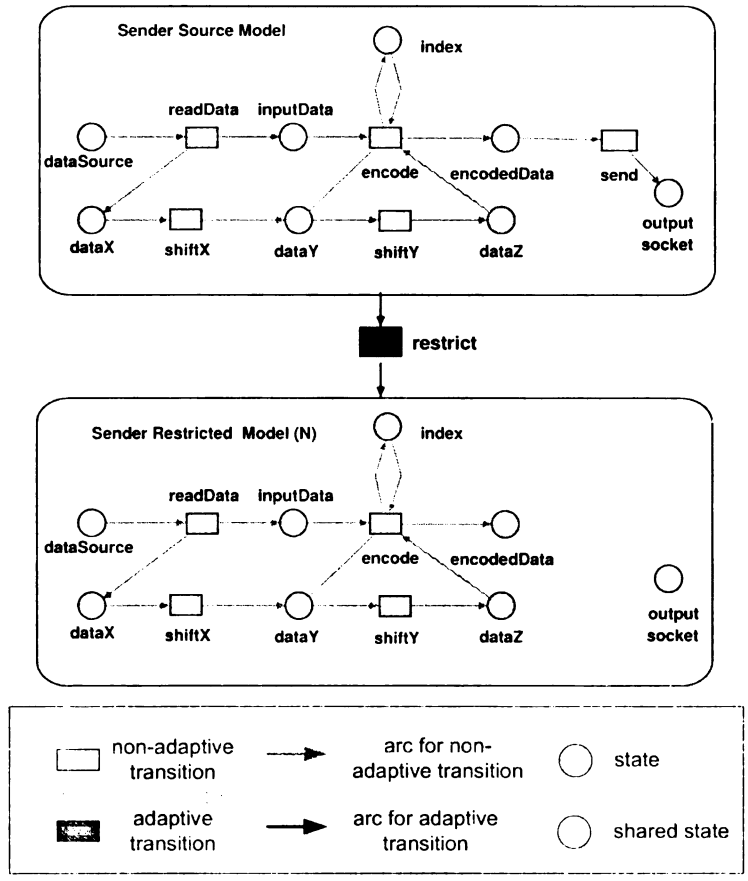


Figure 5.12: Sender restricted source net

constraint: **Once the adaptation starts, it must eventually 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.

To facilitate understanding, we have described the models for the GSM-oriented adaptive communication protocol in a modular fashion, where each discussion focuses on one of the models relevant to achieving adaptation. In fact, the models in Figures 5.7, 5.10, 5.11, and 5.12 can be connected to form a comprehensive model for the GSM-oriented adaptive communication exhibiting overlap adaptation semantics. The

sender and the receiver models are connected by the lossy network net (in Figure 5.7). The adaptation starts when the sender **restrict** transition is fired (in Figure 5.12), and ends when the receiver **adapt** transition is fired (in Figure 5.11). After the sender has adapted to the target (in Figure 5.10) and before the receiver adapts to the target (in Figure 5.11), the source and the target overlap, i.e., the sender exhibits the target sender behavior and the receiver exhibits the source receiver behavior. By considering the sender and the receiver as an entire program, the adaptive GSM-oriented adaptive communication protocol is an overlap adaptation. The constraints for the entire adaptive program are specified as follows:

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

$$(\Box lossCount <= 2) \rightarrow (\Box \neg lose(x)) \tag{5.8}$$

We used model checking to verify this property successfully.

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

$$\Box(senderAdapted \rightarrow \Diamond receiverAdapted). \tag{5.9}$$

We found errors when model checking the adaptation integrity constraint (Formula 5.9). 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 the sender adapts to the target, so that the receiver will be guaranteed to receive the packet. Note that it is generally possible to build a reliable communication channel atop unreliable underlying infrastructure by using acknowledgement-based protocols [129]. Using it to send audio streams would incur a performance penalty. However, if we use it to send only critical packets occasionally, then the penalty is negligible. We repeated the model checking for the revised model against the adaptation integrity constraint (Formula 5.9), and the result showed that the adaptation indeed ran to completion with the revised model, indicating that the property is satisfied.

Adaptation controller net

In a complex adaptation scenario, a number of concurrent adaptive components are involved in a single overlap adaptation process. In order to drive a complex adaptation process, we use an adaptation controller net to model the sequence of adaptive transitions in the adaptation. (In essence, the adaptation controller net serves as a driver for the adaptation process.) Figure 5.13 shows the controller net for the adaptation of the sender and the receiver. The adaptation includes four phases: in source, restricted source, sender adapted, and in target, each of which corresponds to a place in the adaptation controller net. The transitions in the controller net, including restrict, sender adapt, and receiver adapt, correspond to the adaptive transitions in the sender and the receiver adaptation nets.

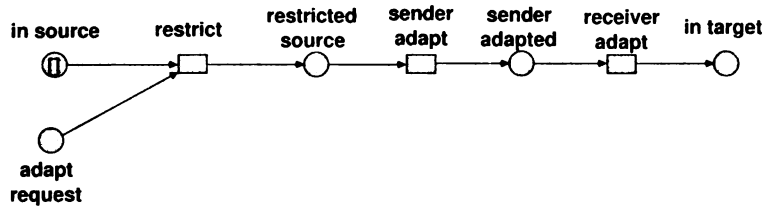


Figure 5.13: Adaptation controller net

As shown in Figure 5.14, we compose the adaptation controller net with the

sender and the receiver adaptation nets to drive their adaptive transitions, thus providing a global view of the adaptation procedure. In this model, the arc connecting the in source place of the adaptation controller and the send transition of the sender source model indicates that the transition is enabled only in the in source phase of the adaptation. Firing the restrict source transition disables the send transition in the sender source model and enables the adapt transition in the sender adaptation model. Firing the sender adapt transition not only transfers the sender state from the source to the target, but also transitions the adaptation phase from restricted source to sender adapted, thus enabling the receiver adapt transition in the receiver adaptation model. Finally, firing the receiver adapt transition moves the adaptation to the in target phase, thus completing the adaptation.

5.3 Reifying the Models

This section introduces 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 [78].

5.3.1 Rapid Prototyping

Renew [78] 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. In model-driven approaches, the model drives 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 method 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 method, which implements

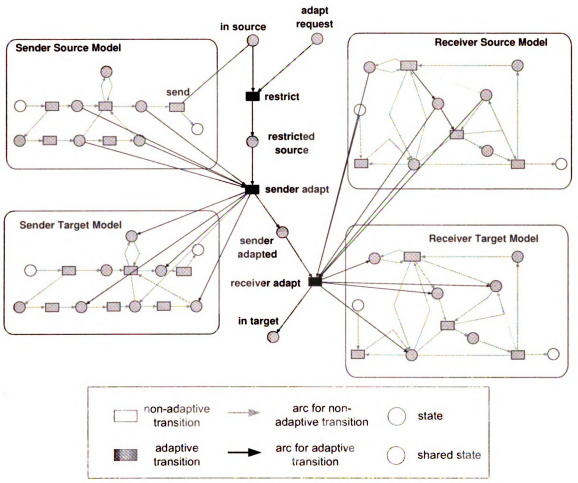


Figure 5.14: Overall adaptation with controller net

the necessary state transformation.

Following the procedure introduced above, we have built a rapid prototype for the adaptive sender and adaptive receiver in Java, and executed the application. The prototype can be included as a module in a Java program to further validate its design. We have tested its execution results, and the rapid prototype executed as expected, thus serving to validate the Petri net models. The detailed Java implementation for rapid prototyping is included in Appendix B.

5.3.2 Model-Based Testing

Given the rapid prototype generated, the production-level adaptive program can be designed and implemented. After the program is implemented, we ensure that the program is implemented properly with a model-based testing technique [112], which is supported by Renew [79]. We verify the following two constraints:

1. Each transition in the model must have a corresponding *handler method* in the implementation, where a handler method for a transition triggers the transition when the method is invoked.

2. The sequence in which the handler methods are called must conform to an allowable transition sequence (i.e., an execution) of the Petri net models.

For each Petri net model, we manually create a *stub* file (in a Renew-specific format), which defines the mapping between handler methods and transitions in the model. For example, we create a file named "`SenderNet.stub`" to define the handler methods for the sender adaptation net in Figure 5.10. The code excerpt from "`SenderNet.stub`" is shown in Figure 5.15. In the stub, line 2 declares that a Java class `SenderNet` is to be created for the sender net. The body of the stub (lines 4–21) defines the mapping between the methods (i.e., handler methods) of the `SenderNet` class and the transitions of the sender net. For example, lines 4–6 define that the `readdata` method of the `SenderNet` class is mapped to the **readData** transition in the net. The stubs for other nets are created similarly. The detailed implementation of other stub files is included in Appendix B.

These stubs are processed by a Renew stub compiler, *compilestub*, which creates Java source code for the handler methods. These methods interact with the Petri net models at run time: Invocations of these handler methods have the following effects.

- If the corresponding transition is enabled at the time a method is invoked, then the transition will be fired and the method will return immediately.

```
01 package gsm;
02 class SenderNet for net sender
03 {
04   void readdata(){
05     this:readData();
06   }
07
08   void S_encode(){
09     this:S_encode();
10   }
11
12   ......
13
14   void T_encode(){
15     this:T_encode();
16   }
17
18   void adapt(){
19     this:adapt();
20   }
21   ......
22 }
```

Figure 5.15: Code excerpt from the sender net stub (`SenderNet.stub`)

- If the corresponding transition is not enabled at the time the method is invoked, then the method will block until the transition is enabled.

For each transition T in the Petri net models, we first manually identify the code segment C (usually a method call) in the adaptive Java program that is intended to implement the transition. We then insert an invocation to the handler method for T at the entry point of C, i.e., at the entry point of the method. During an execution of the Java program, if the sequence in which these handler methods 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 evaluate the conformance between the executions of the Java implementation and that of the Petri net models.

5.4 Case Study: Adaptive Java Pipeline Program

In order to validate the model-based adaptive software development process, we next describe another example application of our approach; specifically, we applied it to the development of an adaptive Java pipeline program [151]. In some multi-threaded Java programs, including proxy servers, data are processed and transmitted from one thread to another in a pipelined fashion. The Java pipeline is implemented using a piped I/O class. The synchronization between the input and the output to the pipeline is implemented using Java `synchronized` functions. Previously, we have studied optimization techniques and proposed an asynchronous Java pipeline design to be run on a multi-processor machine [151]. By eliminating synchronization overhead, the asynchronous version improves performance with a speed up rate of 4.83 over the synchronized implementation when the CPU workload is low. However, when the CPU workload is high, the synchronized version performs better. Based on the above observations, we would like to build an adaptive version of the Java pipeline where the program can choose to use the optimal implementations at run time based on the CPU workload. However, the complexity of designing an efficient and verifiably correct algorithm that switches between synchronization protocols prevented us from implementing the adaptive version of the program using traditional development techniques. With the approach introduced in this chapter, we are able to not only build an adaptive program model, but also model check critical properties, and therefore, gain confidence in the design of the adaptive program.

Applying techniques in Chapter 4, we use a goal model to analyze the adaptation requirements of the adaptive program. As shown in Figure 5.16, the top-level goal G for the adaptive Java pipeline program is to enable efficient pipelined data transmission between a writer thread and a reader thread in Java. Next we refine the goal model.

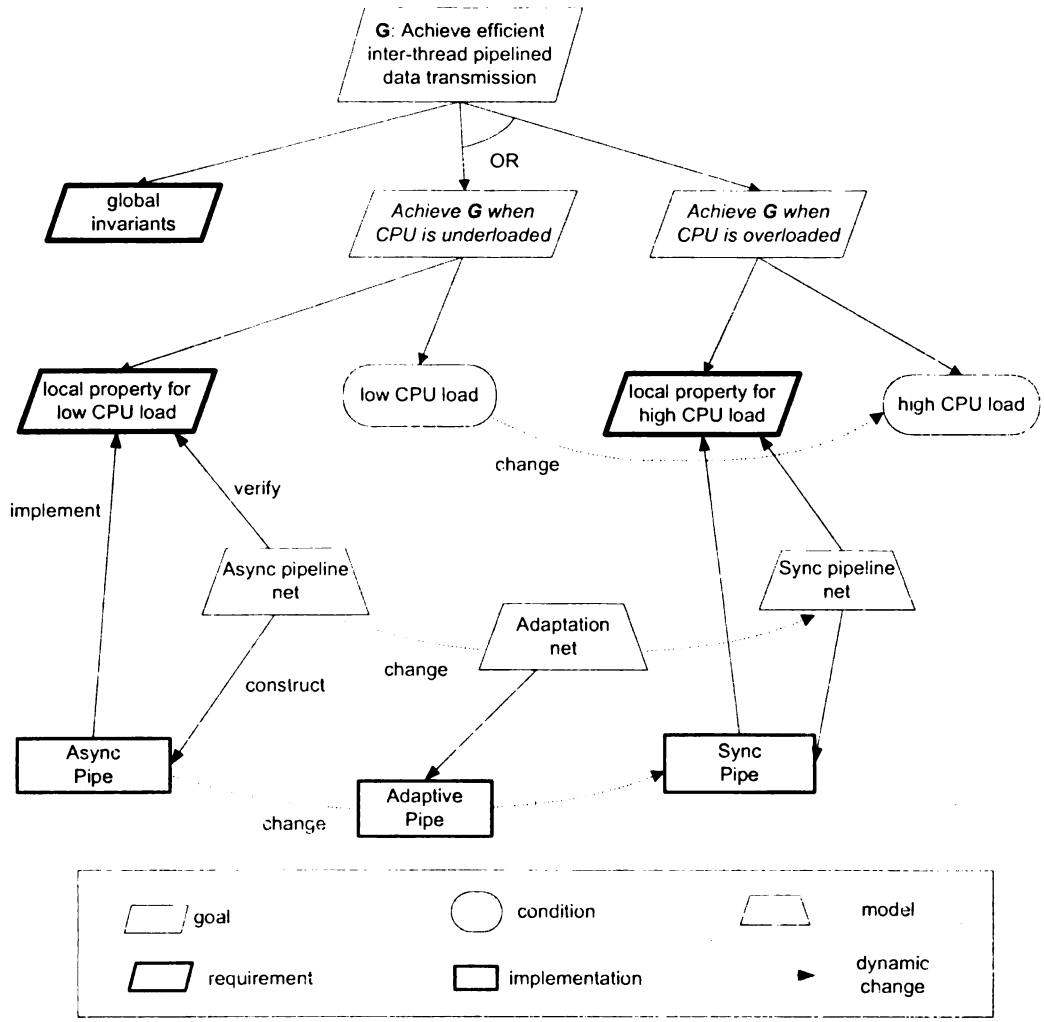


Figure 5.16: Goal model for adaptive Java pipeline

5.4.1 Specifying Global Invariants

We first use global invariants to specify the expected behavior of the Java pipeline program. The set of global invariants serve as a contract between the adaptive pipeline program and its clients, and therefore, must hold regardless of adaptations. The global invariants are as follows:

- **No deadlock-safety invariant**: The program should never deadlock, i.e., the pipeline should continue to read from the input data buffer until the input buffer becomes empty. Or simply stated, the input buffer must finally become empty.

In LTL,

$$\Diamond input\_empty. \tag{5.10}$$

- **No data loss-liveness invariant**: All data input to the pipeline must eventually be output from the pipeline. In LTL,

$$\Box(input(x) \Rightarrow \Diamond output(x)). \tag{5.11}$$

- **No erroneous output-invariant**: The pipeline must not output any data that is not a part of the input data. In LTL,

$$(\neg output(x)\, \mathcal{U}\, input(x)) \vee \Box \neg input(x). \tag{5.12}$$

- **Data ordering-invariant**: The order in which data are read from the pipeline must match the order in which data are written to the pipeline, i.e., no out-of-order transmission. In LTL,

$$(\neg input(x)\, \mathcal{U}\, input(y)) \Rightarrow (\neg output(x)\, \mathcal{U}\, output(y)). \tag{5.13}$$

As shown in Figure 5.16, we create a requirement node under the top-level goal to include these global invariants in the goal model.

5.4.2 Specifying Local Properties

Next, we specify local properties for the Java pipeline program. The adaptive pipeline program executes in a high CPU workload domain and a low CPU workload domain. In the high CPU workload domain, we require the program to operate in the synchronized mode, which has lower concurrency level, but is less CPU intensive. The writer and the reader are not allowed to operate simultaneously. In addition to the global invariants, we have the following local property for this domain:

- **Mutual exclusion-local property**: The read and the write operations must not be enabled simultaneously at any time. In LTL,

$$\Box\neg(write\_enabled \wedge read\_enabled). \tag{5.14}$$

In the low CPU workload domain, we require the program to operate in the asynchronous mode, which is more CPU intensive, but has higher concurrency. In this mode, we require maximal concurrency, i.e., when the buffer is not full (or empty), the write (or read) operation must be enabled.

- **Writer concurrency-local property**: The write operation must be enabled when the buffer is not full. In LTL,

$$\Box(\neg full \Rightarrow write\_enabled). \tag{5.15}$$

- **Reader concurrency-local property**: The read operation must be enabled when the buffer is not empty. In LTL,

$$\Box(\neg empty \Rightarrow read\_enabled). \quad (5.16)$$

As shown in Figure 5.16, we create an OR-refined subgoal for each domain in the goal model. Under each subgoal, we create a requirement node to include local properties and a condition node to include domain conditions.

5.4.3 Constructing Steady-State Models

In this step, we construct steady-state models in Petri nets and verify these models against global invariants and local properties. We build Petri net models for the synchronized and asynchronous pipeline classes. Figure 5.17 shows the Petri net model for the synchronized pipeline class with three internal data buffers buffer1, buffer2, and buffer3, each of which can hold 0 or 1 data token. The three write transitions model the *write* operation that puts input data in one of the empty internal buffers. The three read transitions model the *read* operation that reads data from one of the internal buffers. The places free, reading, and writing model the states in which a *read/write lock* is free, held by the reader, and held by the writer, respectively. A *lock* token must be in the reading (or the writing) place to enable the read (or the write) transitions. The lock can be transferred among reading, writing, and free by the firing one of the reader_lock, writer_lock, write, and read transitions. The input and output places are respectively the data source and the data sink for the model. In the synchronized pipeline model, the write and read transitions must be explicitly enabled by writer_lock and reader_lock, and only one of them can be enabled at any given point in time. We perform model checking to verify that the model satisfies the mutual exclusion local property (formula (5.14)) and all the global invariants (formulae (5.10)-(5.13)). The model checking results showed that these properties

are in fact satisfied by the model.

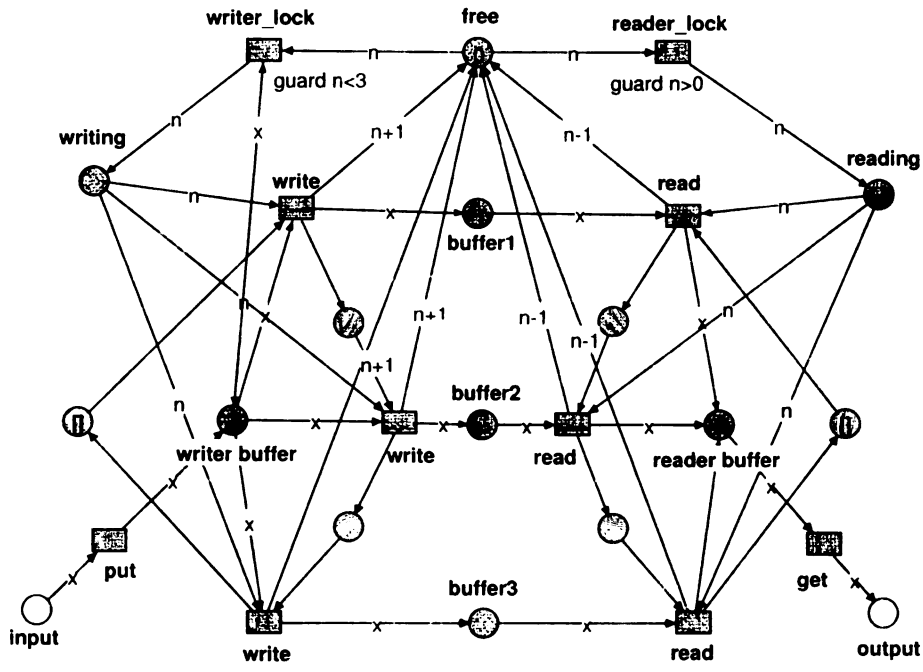


Figure 5.17: Synchronized pipeline net

Next, we build the Petri net model for the asynchronous pipeline program (shown in Figure 5.18). In the asynchronous model, there is no lock controlling the synchronization between the write and the read transitions. Since it allows the writer and the reader to operate the buffers simultaneously, the model must be carefully designed to avoid potential risk of race conditions. Under no circumstances should the reader and the writer both operate the same buffer unit at any given point of time. Model checking has been performed to ensure that the model satisfies the writer and reader concurrency local properties (formulae (5.15), (5.16)) and the global invariants (formulae (5.10)-(5.13)).

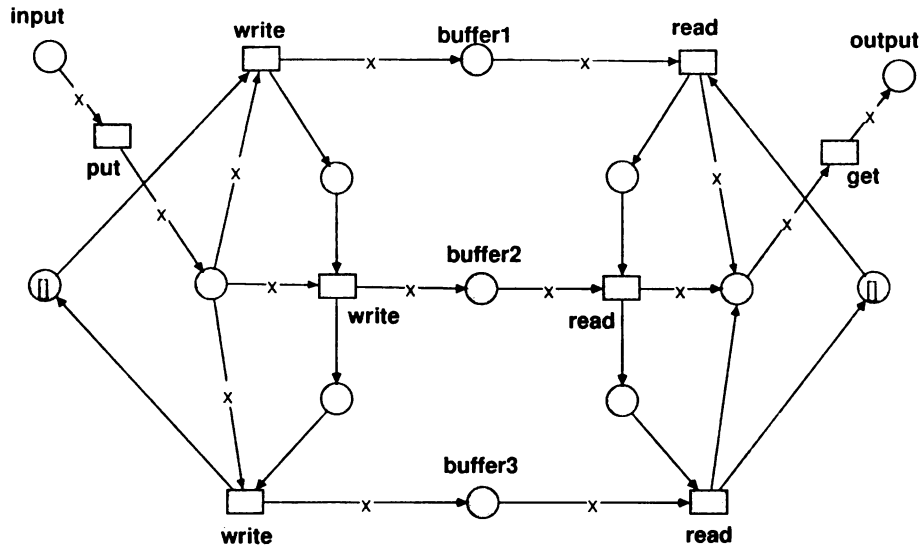


Figure 5.18: Asynchronous pipeline net

5.4.4 Constructing Adaptation Models

In this step, we construct adaptation models in Petri nets and verify these models against global invariants. In Figure 5.19, we show the adaptation controller net for the overlap adaptation from the synchronized pipeline to the asynchronous pipeline. The adaptation process includes three phases: in source, writer adapted, and in target. The first adaptive transition, adapt writer, identifies the quiescent states to be "when the writer is ready to write a data unit to the pipeline". Firing the transition has the following effects:

1. Disable the input to the synchronized pipeline and enable the input to the asynchronous pipeline.

2. Transfer the current input data from the synchronized pipeline to the asynchronous pipeline.

3. Move the current phase to writer adapted.

After the **adapt writer** transition is fired, the writer will start to write to the asynchronous (target) pipeline, and the reader will still read from the synchronized (source) pipeline if there is any data remaining in the buffers. The second adaptive transition is the **adapt reader** transition, which identifies the quiescent states to be "when the buffers in the synchronized pipeline are all empty". Firing this transition will switch the output from the synchronized pipeline to the asynchronous pipeline, and move the current phase to **in target**, thus completing the adaptation process. For the adaptation model, we specify the following adaptation integrity constraint:

- **Adaptation integrity constraint**: If the **adapt writer** transition is fired, then eventually the **adapt reader** transition must also be fired. In LTL,

$$\Box(adapt\_writer \Rightarrow \Diamond adapt\_reader). \tag{5.17}$$

We then model check the adaptation model against the global invariants. During the verification, we found the *no deadlock safety invariant* (i.e., $\Diamond input\_empty$) was violated by the model. By examining the violation path, we found that when the synchronized (source) pipeline was in the **writing** state and when there was input data available in **writer buffer**, firing the **adapt writer** transition moves the input data to the asynchronous (target) pipeline, leaving the synchronized (source) pipeline locked in the **writing** state, thus violating property (5.17). We modified the model by changing the quiescent states for **adapt writer** to be "when **writer buffer** is empty". Model checking results indicated that the new model satisfied all the global invariants.

The models and the analysis capabilities enabled us to focus our attention on the design of the adaptation algorithm in order to maximize the program performance during adaptation. The design of the models has the following performance advantages. First, it is an overlap adaptation, i.e., the writer adapts to the target model

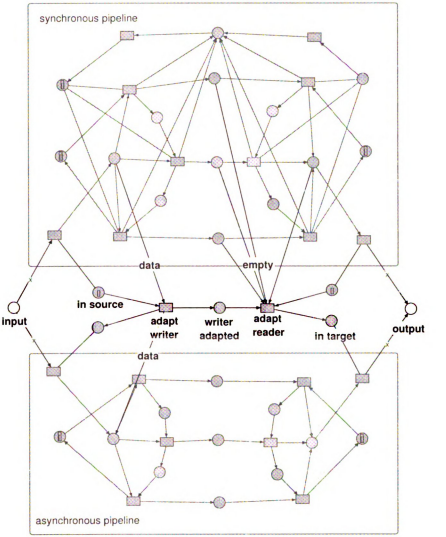


Figure 5.19: Adaptive pipeline adaptation net

without blocking or flushing the buffers in the pipeline. Second, it minimizes the overhead of state transfer, i.e., it does not need to copy data from the source model to the target model.

5.4.5 Reifying the Models

In this section, we demonstrate how the Petri net models created above can be used in the development of an adaptive program. We followed the Petri net

models to build an adaptive program in Java that switches between synchronization protocols under adaptation requests. We built two Java-pipelined objects that implement the synchronized and the asynchronous models, respectively. We created an adaptation driver object that implements the adaptation controller net which sets its states based on the current phase in adaptation. We finally created an adaptive Java-pipelined object that comprises the adaptation driver and the synchronized/asynchronous pipelined objects. The adaptive pipelined object delegates read/write requests to the appropriate pipelined object based on the state of the adaptation driver object.

5.5 Related Work

The work presented in this chapter has been significantly influenced by several related projects on formally specifying adaptive program behavior with process algebraic languages. For example, Kramer and Magee [72] 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* (specified in FSP) 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 passive, and all communications 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 the use of model checking to verify adaptation correctness. Allen *et al* [2] integrated the specifications for both the architectural and the behavioral aspects of dynamically adaptive programs using the Wright ADL. They used 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, thereby achieving separation of concerns. Wright specifications can be converted into a process algebraic language, CSP [58], which can then be statically verified. Canal *et al* [19] used LEDA, an ADL that supports inheritance and dynamic reconfiguration, to specify dynamic programs. LEDA is based on the π-calculus [100], a simple and 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 [2]. It is also possible to derive prototypes and interfaces from the specification automatically.

Below are some of the key advantages when comparing our approach to the above approaches: (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 context 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 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 discussed by all three approaches are simple actions rather than the coordination among concurrent adaptive actions. (5) The above techniques are specific to the type of formalism being used (process algebra), thus making them potentially difficult to be generalized to other types of formalisms.

Theorem proving has also been used to ensure critical properties in adaptive software. Kulkarni *et al* [76] introduced a *transitional-invariant lattice* technique that uses theorem proving to show that during and after an adaptation, the adaptive system is always in correct states with respect to satisfying a set of invariants. A transitional-invariant lattice is a single-source, single-sink directed acyclic graph (DAG), where the source node s represents the source program and the sink node

t represents the target program. An intermediate node n between s and t represents an intermediate program obtained from s by performing one or more adaptive actions. Let $p(n)$ be the program represented by node n, where n can be s, t, or any intermediate node. They associate s and t with the invariants $inv(s)$ for $p(s)$ and $inv(t)$ for $p(t)$, respectively. They also associate each intermediate node n with a transitional-invariant $inv(n)$ for $p(n)$. They prove that, in the lattice, if there is an edge from node n_i to node n_j, then $p(n_i)$ satisfies $inv(n_i)$ implies $p(n_j)$ satisfies $inv(n_j)$. As such, if the source program satisfies its invariants, then they conclude that the target program also satisfies its invariants. There are two major differences between our approaches. First, theorem proving is fundamentally different from model checking in that although it is more powerful in handling certain cases, it requires extensive human intervention during the process. Second, the properties that they discuss are restricted to propositional properties, while we handle both propositional and temporal properties.

5.6 Discussion

In this chapter, we have proposed a model-driven software development process for creating state-based models for adaptive software 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.

The concept of quiescent states is similar to fulfillment states introduced in Chapter 3, where a fulfillment state is defined to be a state in which all the obligations of the source behavior are fulfilled, and thus making it safe to terminate the source behavior. The "fulfillment state" is a concept at the requirements level, while "quiescent state" is a concept at the design level. A quiescent state may or may not be a fulfillment state depending on the effect of the adaptation: If its effect is to terminate

the source behavior, such as the adapt transition in Figure 5.10, then it must be a fulfillment state; otherwise, such as the restrict transition in Figure 5.12, it may not be a fulfillment state.

Our experience shows that the proposed approach has the potential to improve both the development time and reliability of the code. The running adaptive audio streaming example had been originally developed by members in our research lab without the proposed approach. After applying our approach to the example, the original developers found that our approach significantly improved their understanding of the problem, and the new design was clearer than the original one. Our approach reduced the developer's burden by separating different concerns and leveraging automated model construction and analysis tools. In the adaptive Java pipeline example, the proposed approach helped us focus on the design of adaptation algorithms, making it possible for us to build an efficient and verifiably correct adaptive Java pipeline program. Automated analysis has played an essential role by reducing a significant portion of burden of verifying the correctness of the adaptation.

The discussions in this chapter have focused on the development of new adaptive software. We also investigated the application of the model-based technique to enabling dynamic adaptation in legacy software with assurance, which will be discussed in Chapter 6.

In this chapter, we focused our discussion on the verification of global invariants and local properties specified in LTL. These properties are supported by existing tools, such as MARIA. The verification of adaptation variant properties specified in A-LTL will be described in Chapter 7.

Chapter 6

Re-Engineering Software to Enable Adaptation

In this chapter, we extend the MASD approach introduced in Chapter 5 to enable dynamic adaptation in legacy non-adaptive software [149]. We focus on the connection between design models and implementations of adaptive software.

There is a gap between existing modeling and implementation-based approaches for adaptation: On the one hand, existing techniques [1, 12, 15, 43, 69, 114, 122, 126] that address 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, existing techniques [2, 18, 19, 57, 72, 73, 99, 106, 128, 131] addressing correctness in dynamic adaptation using rigorous software engineering techniques focus on abstract models and do not take the models to their implementations.

In order to bridge the gap between existing modeling and implementation-based approaches for adaptation, we introduce a formal technique to ensure that the adaptation requirements for the software are satisfied by the adaptive software implementation. The key insight is that the Unified Model Language (UML) models, with

formally defined semantics [97], can be used as an intermediate representation to bridge the gap between adaptive software implementations and the formal models used for adaptive software verification. the design for adaptation can be performed on the UML models by creating adaptation UML models, which can be automatically translated into formal models using existing tools [97] for formal analysis.

Our approach has three key dimensions: model driven, non-invasive to the legacy code, and enables formal analysis of the resulting adaptive systems for adherence to local invariants, and adaptation properties. We leverage and extend several key enabling techniques that we and others developed previously. First we leverage the MASK approach in Chapter 5 to gain a model-driver approach Second, we extend an aspect-oriented adaptation enabling technique [139] to enable non-invasiveness to the legacy code. Third we leverage and extend a UML formalization framework [97] to reverse-engineer legacy code to UML models and formalize UML models. The aspect-oriented adaptation enabling technique allows us to insert adaptation related code to existing software without directly altering original legacy source code, i.e., it is non-invasive. Non-invasiveness to the source code is important in order to enable the adaptation code and the legacy code to be maintained separately. The assurance in this approach is achieved by model checking the design models and support for systematic translations between the models and their implementations. Furthermore, our approach allows more flexible adaptation when compared to existing adaptation enabling techniques [115, 121, 139]. Many adaptation techniques addressing legacy code [115, 121, 139] require the points for adaptation to be separated from the code segment changed by the adaptation. As discussed in section 5.2.3, this constraint may impose unacceptable performance penalties in some adaptation scenarios. Thus, we introduce a *cascade adaptation technique* that enables more flexible adaptation, thus allowing adaptation to start at more points in the program.

We have applied our approach to the adaptive Java pipeline program [151] intro-

duced in Chapter 5. We demonstrate our approach by re-engineering a non-adaptive version of the Java pipeline program to become an adaptive version using our proposed approach. The remainder of this chapter is organized as follows. Section 6.1 overviews earlier work on an aspect-oriented adaptation enabling technique [139] and a metamodel-based UML formalization technique [97]. In Section 6.2, we describe a simplified version of our approach to be used in scenarios where there is only one adaptive component adapting to only one target behavior. A case study is described in Section 6.3. Section 6.4 describes a number of extensions to the simplified technique for more general adaptation scenarios. Section 6.6 summarizes this chapter and discusses limitations and extensions of our approach.

6.1 Background

In this section, we overview two techniques that are extensively leveraged for our proposed approach. The aspect-oriented adaptation enabling technique [132] is leveraged to enable non-invasive insertion of adaptation code into the legacy software. The metamodel-based UML formalization technique [97] is used for two purposes: reverse-engineer legacy software and formalize UML models.

6.1.1 Aspect-Oriented Adaptation Enabling Technique

Aspect-Oriented Programming (AOP) [67] techniques encapsulate crosscutting concerns of a program into single entities called *aspects*. *AspectJ* [132] is an aspect-oriented extension to the Java programming langauge. In AspectJ, each crosscutting concern is defined in an aspect, stored separately in an aspect file with the file name extension "`.aj`". An aspect definition comprises *pointcuts* and *advices*. A *pointcut* defines a set of points (named *join points*) in a program, such as invocations of a certain function, access to a certain variable, etc. An *advice* comprises the type of

the advice (*before*, *after*, or *around*), a set of pointcuts, and a code segment. A before (resp. after) advice causes the AspectJ compiler to insert the code segment before (resp. after) every join point in the program matched by the pointcuts. An around advice causes the AspectJ compiler to replace the join points matched by the pointcuts with the code segment.

Yang *et al* [139] previously developed a two-phased AOP-based technique to enable adaptation in legacy software using AspectJ. As shown in Figure 6.1, in the first phase, occurring at compile time, an aspect fragment, called *behavior adaptor*, defines the points in a legacy program at which "hooks" 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, occurring at run time, 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. We leverage the aspect-oriented adaptation enabling technique to enable non-invasive insertion of adaptation code in the legacy software.

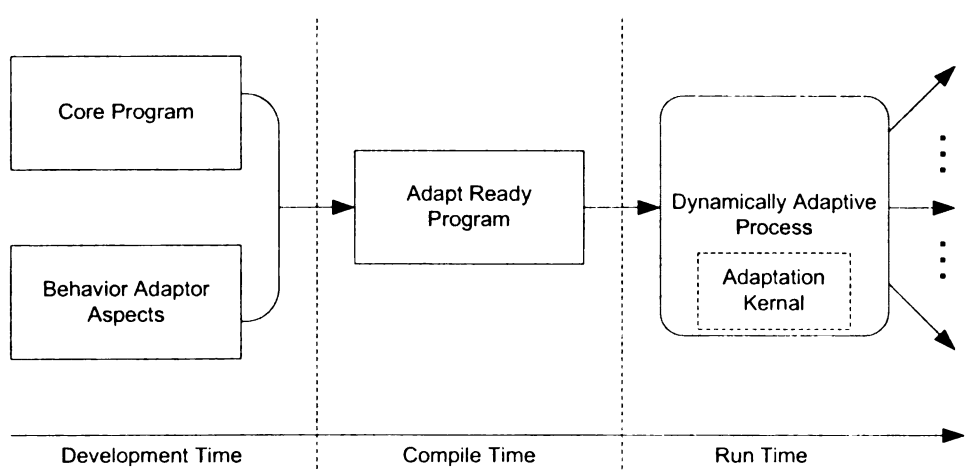


Figure 6.1: Aspect-oriented adaptation enabling

6.1.2 MetaModel-Based UML Formalization Technique

McUmber and Cheng developed a general framework [23] based on mappings between *metamodels* (i.e., class diagrams depicting abstract syntax) for formalizing a subset of UML diagrams in terms of different formal languages, including Promela [97]. 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 [23]. 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 [97]. We use Hydra to translate the UML diagrams created for adaptation to Promela models to be analyzed by the Spin model checker [12].

6.2 Model-Based Re-Engineering

We now show how the aspect-oriented adaptation enabling technique [139], the MASD process, and the metamodel-based formalization technique can be integrated and leveraged to bridge the gap between adaptation implementations and formal models for adaptive systems.

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 6.2, the overall process includes the following four steps: Step (1) occurs in the requirements analysis phase. We perform requirements analysis to elicit a set of global invariants and local properties. Based on these properties, we select from a code base a set of non-adaptive legacy programs $P_1, P_2, \cdots, P_k$, each of which differs from another by one or more segments of code.

Step (2) occurs in the model design and analysis phase. The programs are reverse-engineered into UML Statechart diagrams $M_1, M_2, \cdots, M_k$, named *steady-state models*. These models are then translated into formal models and verified against their local properties using automated tools [97]. Step (3) also occurs in the model design and analysis phase. After the steady-state models are verified, developers must create an *adaptation model* $M_{i,j}$, 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 integrated and translated into adaptive programs. In this step, our approach addresses the following questions: What mechanisms 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?

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. Collaborative adaptive components with multiple potential target programs are discussed in Section 6.4.

6.2.1 Requirements Analysis

In order to re-engineer legacy code, we customize and apply the goal-based requirements analysis for adaptive software introduced in Chapter 4. We use Figure 6.3 to guide our discussion. Assume that the adaptive software is required to achieve a high-level goal G in a set of different domains $D_1, D_2, \cdots, D_k$. The local properties (in LTL) for these domains are $R_1, R_2, \cdots, R_k$, respectively. The set of global invariants (in LTL) for the adaptive software is INV. Also 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

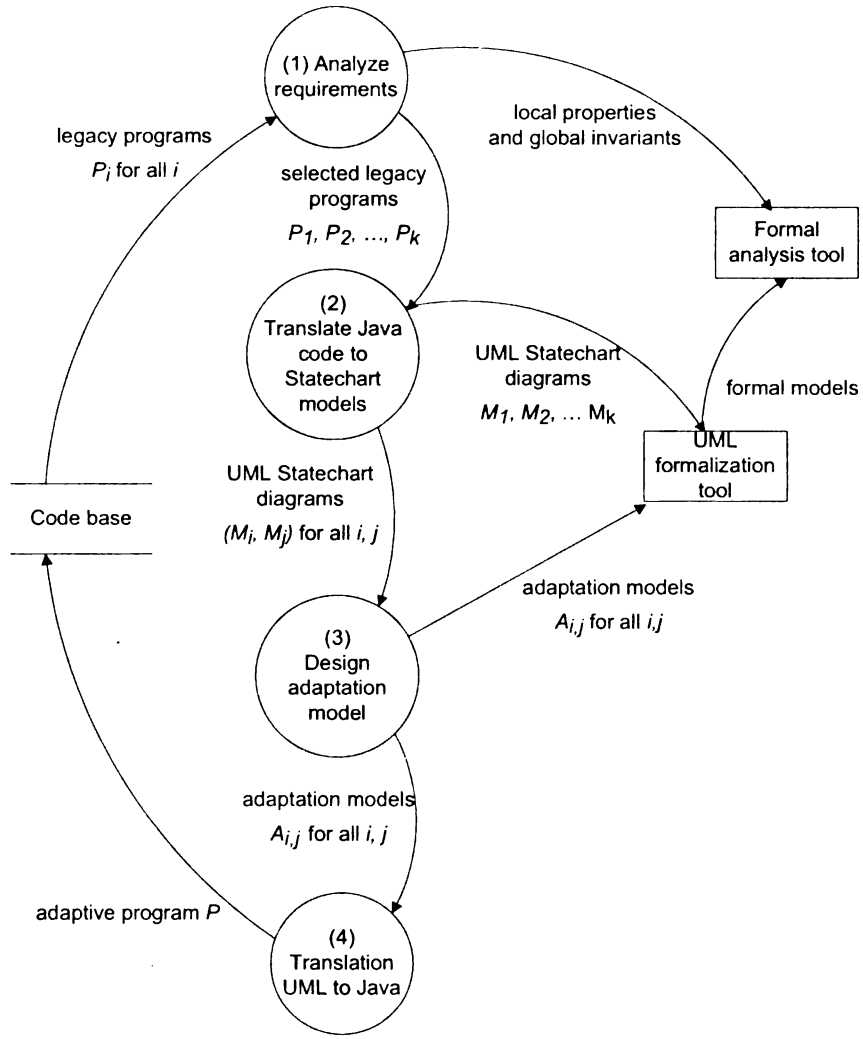


Figure 6.2: Dataflow diagram for the proposed re-engineering approach

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 context, we assume that the legacy programs for all the requirements already exist. Each of these legacy programs will become a steady-state program in the adaptive program.

Next, we determine 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

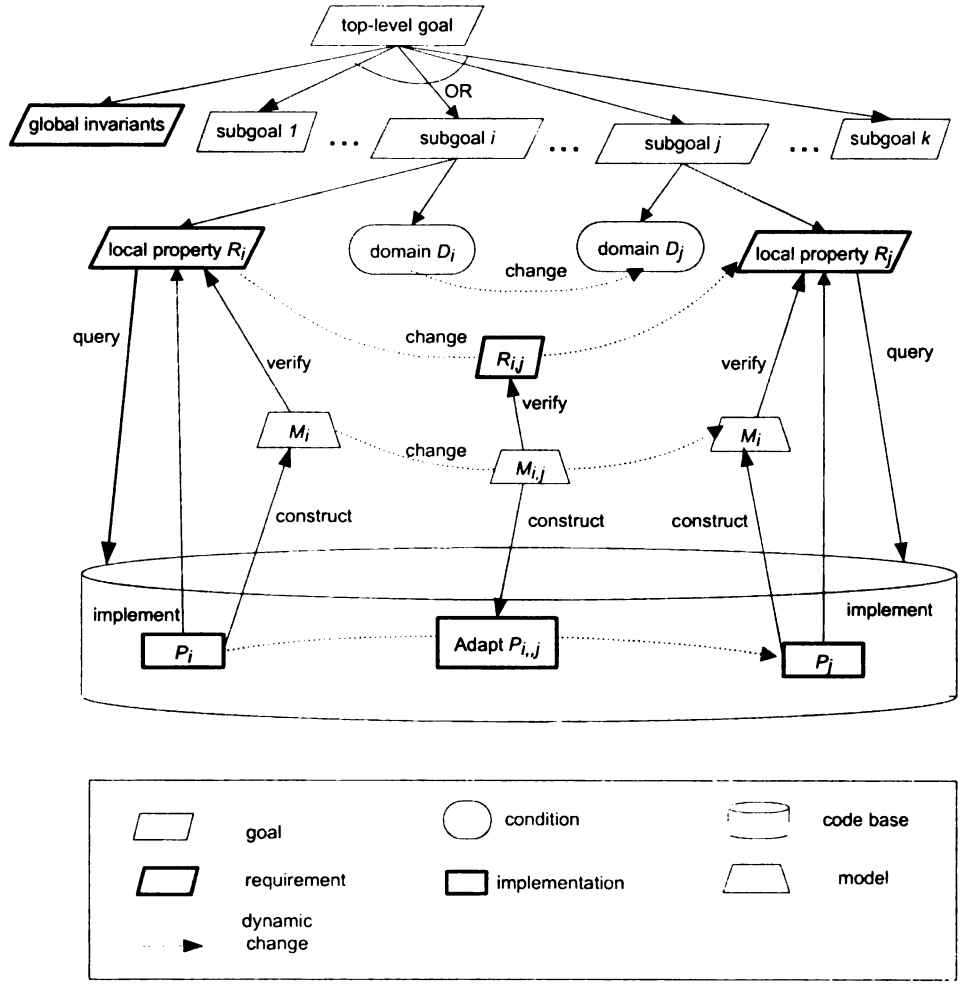


Figure 6.3: Goal model for adaptive software

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.

6.2.2 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 $M_{i,j}$ comprising 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. The above process is very similar to the MASD process introduced in Section 5.2 except that the models M_i are reverse engineered from existing programs rather than refined from requirements as in forward engineering.

Generate Statechart diagrams

We use a metamodel-based technique [97] to generate the Statechart 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 the legacy code, Java in this case, and for UML diagrams. 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. The metamodels and rules are included in Appendix C. Developing the rules for translating the full Java language is non-trivial; ongoing investigations are underway by other groups [68].

Verify local properties for assurance

We use the Hydra tool suite to transform the UML models M_i to Promela models. Then we use the Spin model checker [12] to verify the Promela models against the local properties specified in the requirements analysis. Violations of local properties by the models may indicate one or more of the following cases: (1) The legacy programs P_i

were implemented incorrectly, (2) the UML models M_i have not been generated to actually reflect the legacy programs P_i, or (3) the local properties R_i are specified incorrectly. Then we inspect the above artifacts to determine which one is at fault and the corresponding erroneous artifacts must be revised until the models conform to the properties.

Design adaptation models

After the UML models are generated, we design an adaptation model $M_{i,j}$ 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 $M_{i,j}$ from M_i to M_j by adding adaptive states and transitions to M_i and M_j such that the global invariants are preserved before, during, and after adaptation in $M_{i,j}$. This task is achieved by applying the MASD process introduced in Chapter 5 to Statechart diagrams in three steps: (1) Identify the *quiescent states* in the source model M_i; (2) identify the *entry states* in the target model M_j; (3) determine the state transformation from the quiescent states to the entry states.

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 should 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

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

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 dead (i.e., not used any longer) or 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.

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 model 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.

6.2.3 Code Generation

After the adaptation model is constructed, we implement it 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. Next, we describe each step in further detail.

Construct adaptive classes

We implement adaptive programs by systematically following the adaptation models. Since the UML models were initially generated from the programs, we assume the traceability links between the UML models and the original Java programs are already established in the generation process. Traceability links are the mappings between states in the UML models and statements in the programs. These links can be stored in the UML models as annotations recording the line numbers in the programs. We will introduce a concrete example of traceability links in Section 6.3. 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. 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 location in the target program.

Next we describe a challenge that arises in the implementation of adaptive transitions from quiescent states to entry states and introduce a *cascade adaptation mechanism* to address this challenge.

The Challenge. A simple case of such kind of transitions is illustrated in Figure 6.4, where the code segment to be changed is enclosed by the method `bar()` (lines 1-4), and the quiescent/entry state for adaptation is "outside" of the method `bar()` (line 8). A number of techniques have been proposed to address such types of simple transitions, including the code hot-swapping mechanism [5], the *strategy* design pattern approach [49], etc.

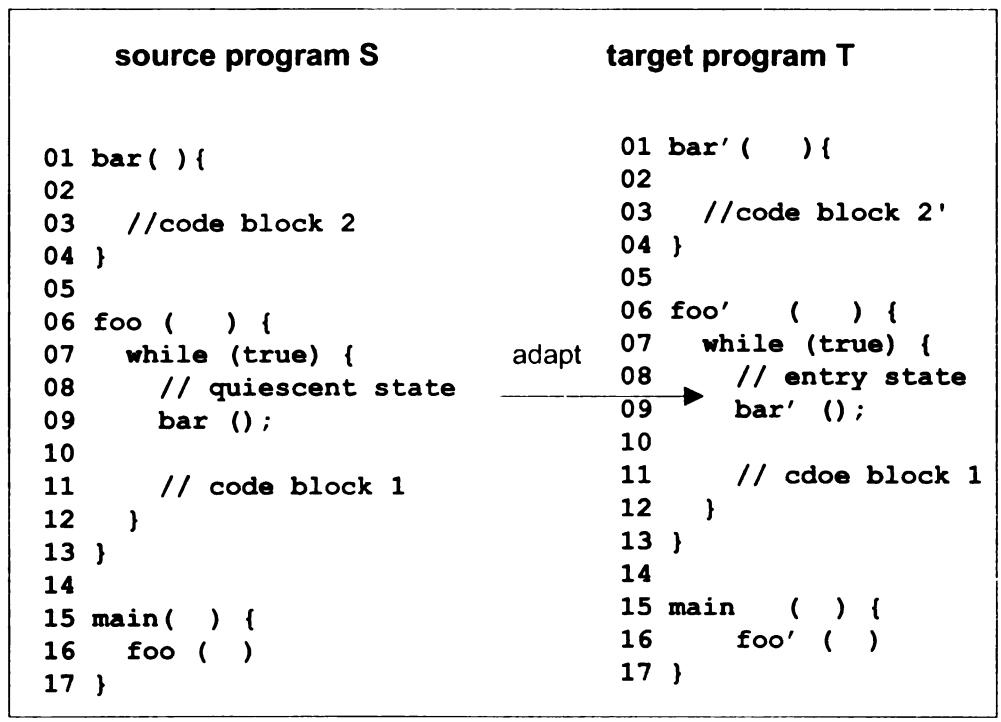


Figure 6.4: An example of the simple case of quiescent/entry states

However, in a more general case, the code segments that are to be changed by the adaptation may be scattered across the source and the target programs. As illustrated in Figure 6.5, the quiescent/entry states are within the code segment changed by the adaptation. We believe that, in legacy code, scattering is the norm rather than the exception, since we have encountered a number of adaptation scenarios of this nature in our research, including the adaptive Java pipeline [145]. The code hot-swapping or the strategy pattern approach will not work in the general case since there is no location in the code where a code hot-swap will have the intended behavior.

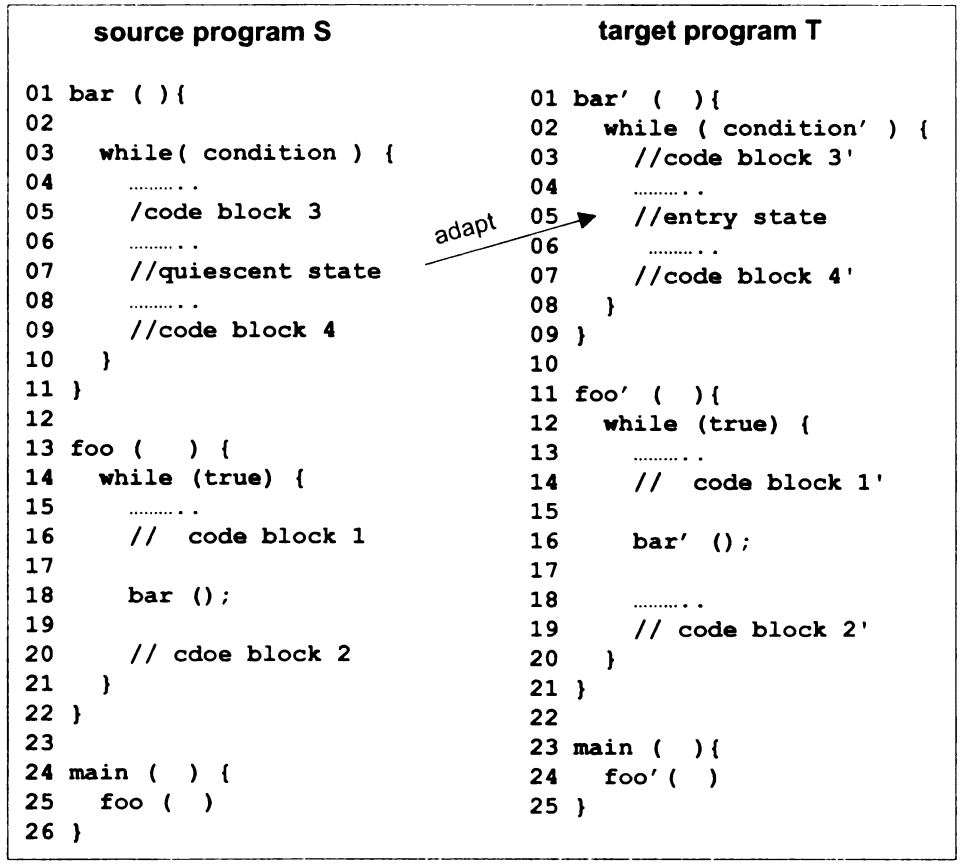


Figure 6.5: An example of the general case of quiescent/entry states

The Cascade Adaptation Mechanism. In order to handle the general case, we propose a *cascade adaptation mechanism* in which the program control first cascades outwards (*outbound cascade*) from the quiescent state in the source program, then cascades inwards (*inbound cascade*) to reach the entry state in the target program. As shown in Figure 6.6, the program control first cascades from the quiescent state (line 7) of the inner method outwards until it reaches a location (line 24) outside the code segment to be changed (lines 1-21). After a state transformation from the source program to the target program, the control cascades inwards until it reaches the entry state of the target program. We implement the outbound cascade using the exception handling mechanism in Java. First, we insert code at the location for the quiescent

state (line 7) to test for adaptation requests. If an adaptation request has been received, then the source program creates an exception object SourceState and stores the local state information of the method `bar()` in SourceState. Then SourceState is thrown to the method `foo()` (line 17), which catches SourceState and incrementally records its own local state information in the object. Finally, the `main()` method catches SourceState (line 24). After transforming SourceState to TargetState, the program control is transferred to the target program.

In order to implement the inbound cascade in the target program, we transform the program into a different target program T' such that its initial state is equivalent to the entry state of the target program. This is achieved by (1) constructing the control flow diagram for the target program, (2) designating the entry state the initial state of the diagram, and (3) regenerating the program from the diagram. When T' is invoked, the program control will jump to the state equivalent to the entry state of the target program. The target state object is then passed inwards as a parameter (line 35 $\rightarrow$ line 18 $\rightarrow$ line 3 in Figure 6.6).

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.

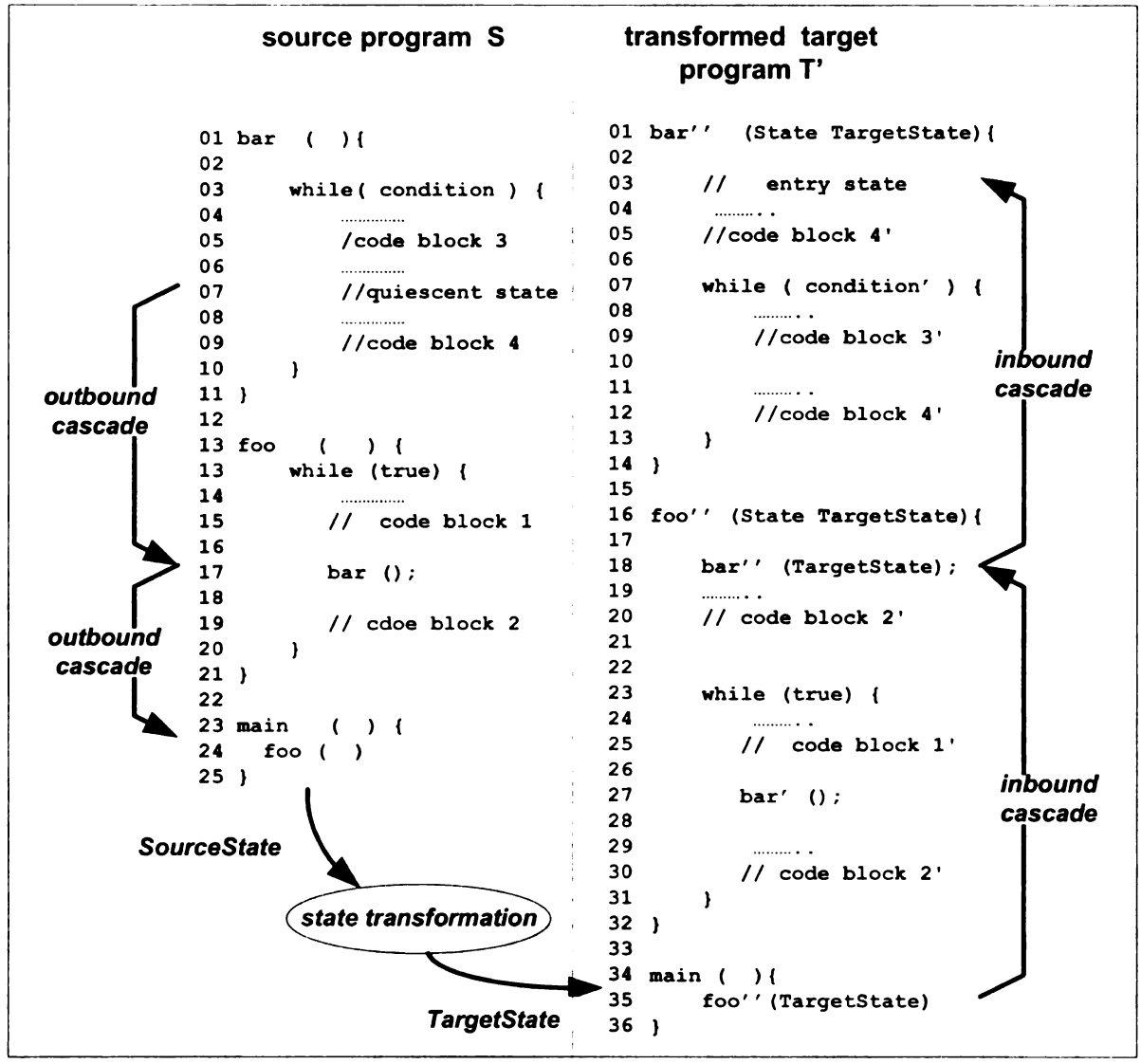


Figure 6.6: The cascade adaptation mechanism

For example, we can define a pointcut FOO that identifies the constructor of a non-adaptive class Foo as follows:

```
public pointcut FOO(): call (Foo.new(...));
```

The following *around advice* definition replaces the constructor of class Foo identified by FOO with the constructor of an adaptive class Bar:

```
FOOBAR around():FOO(){
    return new Bar();
```

}

The objects of the adaptive class (`Bar`) then can be used throughout the legacy program in the same way as for the non-adaptive objects (`Foo`), 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.

6.3 Case Study: Adaptive Java Pipeline Program

In this section, we demonstrate the process of re-engineering legacy programs using the Java pipeline example introduced in Section 5.4.

6.3.1 Requirements Analysis

In Section 5.4, we have introduced the requirements analysis of the Java pipeline program with the MASD approach. The approach introduced in this chapter differs from the MASD approach in the way steady-state models and steady-state programs are obtained. For completeness of the example, we repeat the requirements analysis step here.

We apply the goal-based requirements analysis to the Java pipeline program. The program is required to achieve the high-level goal: To enable efficient pipelined data transmission between a writer thread and a reader thread in Java. We create a top-level goal G in a goal-model as shown in Figure 6.7.

Specify global invariants. We have discussed the global invariants for the adaptive Java pipeline program in Section 5.4. For convenience, we repeat the specifications here. The set of global invariants serve as a contract between the adaptive

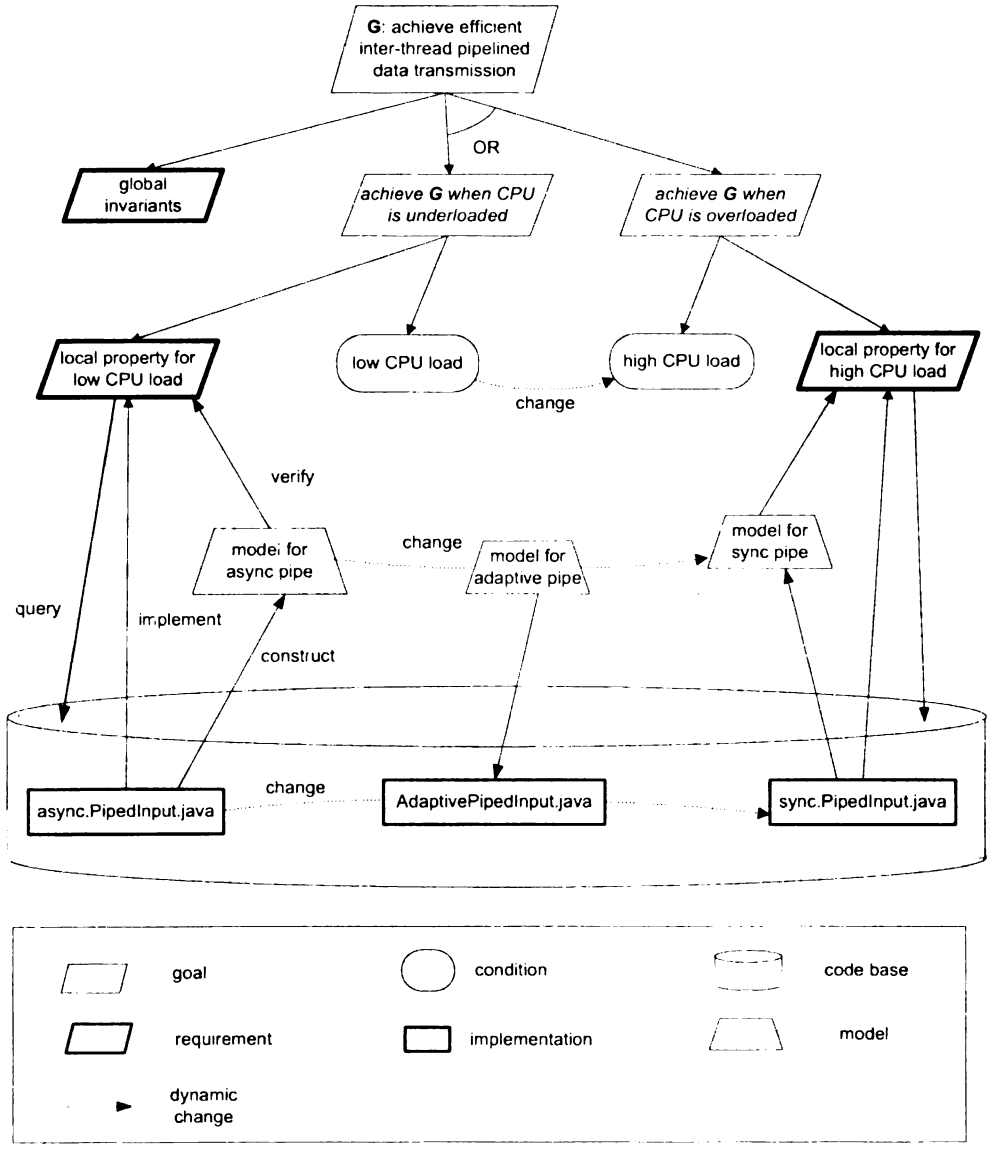


Figure 6.7: Goal model for adaptive Java pipeline

pipeline program and its clients, and therefore, must hold regardless of adaptations. The global invariants are as follows:

- **Safety invariant: no deadlock.** The program must never deadlock, i.e., the pipeline should continue to read from the input data buffer until the input buffer becomes empty. Or simply stated, the input buffer must finally become empty. In LTL,

$$\Diamond input\_empty. \tag{6.1}$$

- **Liveness invariant: no data loss.** All data input to the pipeline must eventually be output from the pipeline. In LTL,

$$\Box(input(x) \Rightarrow \Diamond output(x)). \tag{6.2}$$

- **Invariant: no erroneous output.** The pipeline must not output any data that is not a part of the input data. In LTL,

$$(\neg output(x)\, \mathcal{U}\, input(x)) \vee \Box \neg input(x). \tag{6.3}$$

- **Invariant: data ordering.** The order in which data are read from the pipeline must match the order in which data are written to the pipeline, i.e., no out-of-order transmission. In LTL,

$$(\neg input(x)\,\mathcal{U}\,input(y))\Rightarrow(\neg output(x)\,\mathcal{U}\,output(y)). \tag{6.4}$$

We create a requirement node under the top-level goal to include these global invariants in the goal model.

Specify local properties. Again, we repeat the analysis of the local properties of the adaptive Java pipeline program that we discussed in Section 5.4. The adaptive pipeline program executes in a high CPU workload domain and a low CPU workload domain. In the high CPU workload domain, we require the program to operate in the synchronous mode, which has lower level of concurrency, but is less CPU intensive. The writer and the reader are not allowed to operate simultaneously. In addition to the global invariants, we have the following local property for this domain:

- **Local property: mutual exclusion.** The read and the write operations must not be enabled simultaneously at any time. In LTL,

$$\Box\neg(write\_enabled \wedge read\_enabled). \tag{6.5}$$

In the low CPU workload domain, we require the program to operate in the asynchronous mode, which is more CPU intensive, but has higher concurrency. In this mode, we require maximal concurrency, i.e., when the buffer is not full (or empty), the write (or read) operation must be enabled.

- **Local property: writer concurrency.** The write operation must be enabled when the buffer is not full. In LTL,

$$\Box(\neg full \Rightarrow write\_enabled). \tag{6.6}$$

- **Local property: reader concurrency.** The read operation must be enabled when the buffer is not empty. In LTL,

$$\Box(\neg empty \Rightarrow read\_enabled). \tag{6.7}$$

As shown in Figure 6.7, we create an OR-refined subgoal for each domain in the goal model. Under each subgoal, we create a requirement node to include local properties and a condition node to include domain conditions.

Query the Code Base. In the existing code base, we have two different implementations for the Java pipeline: a synchronized implementation comprising `sync.PipedInput` and `sync.PipeOutput` classes, and an asynchronous implementation comprising `async.PipedInput` and `async.PipedOutput` classes. The descriptions for these two implementations indicate that they are appropriate candidates for the high CPU load domain and the low CPU load domain, respectively.

6.3.2 Design and Analysis

This section describes the model design and analysis for the adaptive Java pipeline example.

Generate state diagrams

We use the metamodel-based approach to translate the Java classes into Statechart diagrams. The piped input and output classes in the legacy program basically implement a `read()` method for input and a `write()` method for output. The implementations of these two methods are largely symmetric. For brevity, we use the `read()` method in the piped input classes (synchronized and asynchronous) as an example to illustrate the procedure. The translation includes two substeps: First, we translate each Java program into an equivalent Java program in a *canonical form*, then we apply the metamodel-based model translation technique to translate the canonical form Java program into a Statechart diagram. The same procedure is applied to the `write()` method and all other methods in the Java classes.

Substep 1: Convert Java code into canonical form. We translate a Java program into an equivalent program in a canonical form in order to simplify the number of different cases we have to handle in the metamodel-based translation.

First, we translate all "`while`" loops and "`for`" loops in the program into "`while(true)`" loops and "`if`" blocks. Figure 6.8 (a) shows the translation of a `while` loop, where the loop body is copied from line 2 in the non-canonical form program to line 3 in the canonical form program. Figure 6.8 (b) shows the translation of a `for` loop, where the loop body is copied from line 2 in the non-canonical form program to line 4 in the canonical form program.

Second, we perform an inline operation to method invocations in the program to *flatten* the syntactic structure of the program. Note that this step currently does not handle recursions.

Figure 6.9 shows the canonical form conversion of the `read()` method of the `sync.PipedInput` class. The method invocation (line 3) in the non-canonical form is inlined in the canonical form (lines 4 21 (L1)). The `while` loop in the non-canonical

```
01 while (condition){
02   [body block]
03 }
```

non-canonical form

```
01 while(true){
02   if (condition){
03     [body block]
04   }else
05     break;
06 }
```

canonical form

(a) `while` loop conversion

```
01 for (x; y; z){
02   [body block]
03 }
```

non-canonical form

```
01 x;
02 while (true){
03   if (y){
04     [body block]
05     z;
06   } else
07       break
08 }
```

canonical form

(b) `for` loop conversion

Figure 6.8: Java code canonical form conversion

form (lines 6–16) is converted to the `while(true)` loop in the canonical form (lines 24–39 (L2)).

Substep 2: Translate Java code into Statechart diagrams. We use the metamodel-based approach to translate the canonical form Java programs into Statechart diagrams. The translation of the `read()` method of the `sync.PipedInput` class is illustrated in Figure 6.10, where Figure 6.10 (a) shows the canonical form Java program that we constructed in the previous substep, and Figure 6.10 (b) shows the corresponding Statechart diagram created with IBM Rational XDE [62]. In the Statechart diagram, each state corresponds to a *point* in the program, where a point is defined to be the location between two lines of code. In Figure 6.10 (a) we have labeled these points with bold comment lines prefixed with "`//state:`". The dotted lines between Figure 6.10 (a) and Figure 6.10 (b) show the traceability links between

Figure 6.9: Before and after canonical form conversion of `sync.PipedInput.read()`

the points in the Java program and the states in the Statechart diagram. We have included more details of the metamodel-based model translation rules in Appendix C.

We generate a Statechart diagram for each of the Java pipeline classes. Every model represents a certain kind of behavior, and thus is considered a steady-state model.

Verify steady-state models

We use the Hydra tool suite to transform the steady-state models to Promela models. Then we use the Spin model checker [12] to verify the Promela models against the local properties specified in Section 6.3.1. In this example, we did not find errors in the models, and therefore, proceed to the next step.

Design adaptation models

Next, we design the Statechart adaptation models. Figure 6.11 shows the adaptation model for the `read()` method of the piped input, where Figure 6.11 (a) and Figure 6.11 (b) represent the synchronized and asynchronous steady-state models, respectively. The bold-lined states and transitions connecting Figure 6.11 (a) and Figure 6.11 (b) represent adaptive states and transitions. The shaded states in the steady-state models are the quiescent state and the entry state.

We chose the state finished? in the source model to be the quiescent state because (1) it is in a loop that is frequently executed, and (2) its condition is relatively simple to specify when compared to other states in the same loop. We chose the state more? in the target model to be the entry state because its position in the target is similar to that of the state finished? in the source, thus simplifying the state transformation.

The adaptation procedure includes three transitions: (1) test the adaptation request and save current state, (2) transform the saved state of the source model to a state of the target model, and (3) restore the state in the target model.

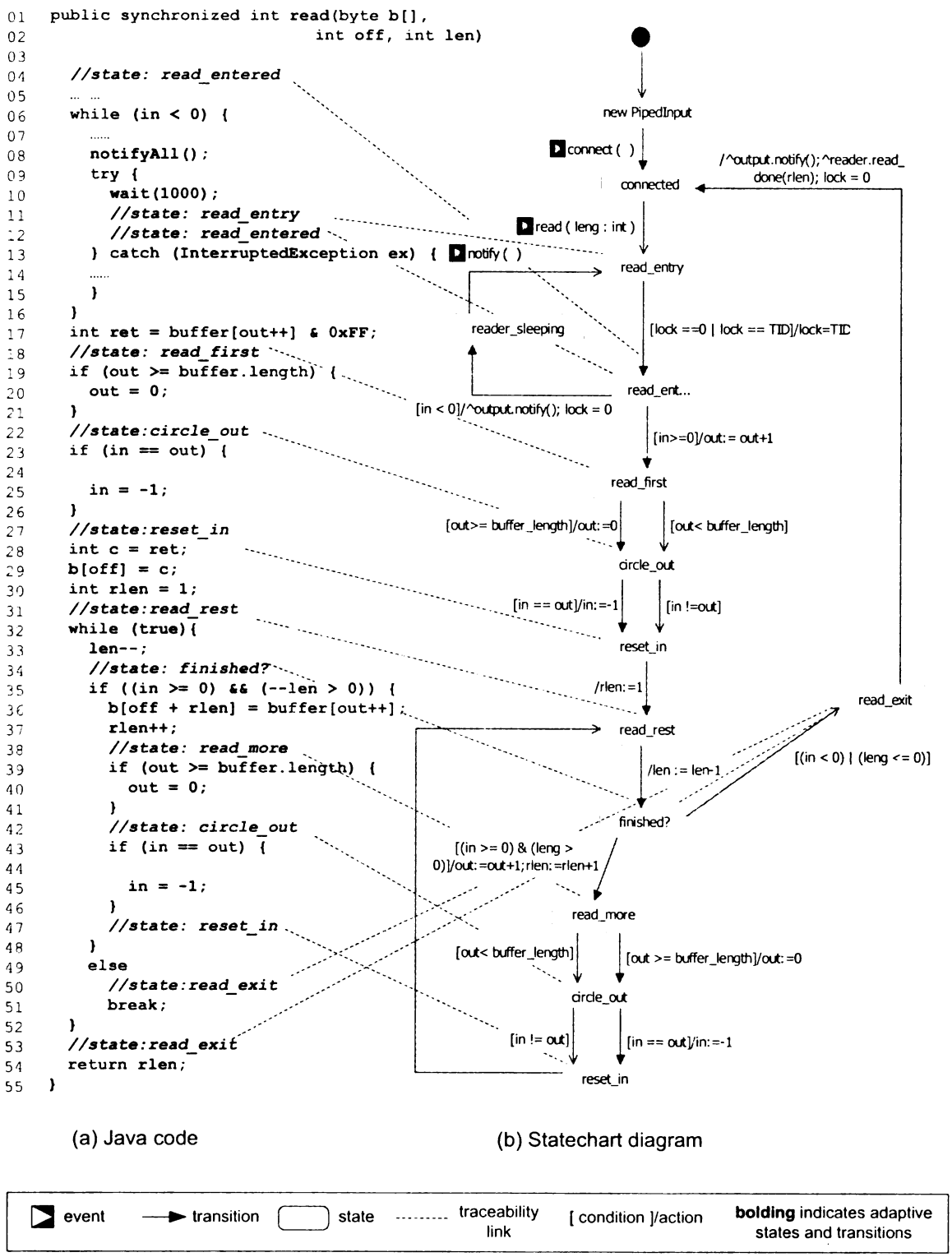


Figure 6.10: Statechart translation of `sync.PipedInput.read()`

The adaptation model for the `write()` method of the piped output can be constructed similarly.

Verify adaptation models

We use the Hydra tool suite to transform the adaptation models to Promela models. Then we use the Spin model checker [12] to verify the Promela models against the global invariants specified in Section 6.3.1. In this example, we did not find errors in the models, and therefore, successfully completed the model design and analysis for the adaptive Java pipeline program.

6.3.3 Code Generation

After the adaptation models are created, we generate Java code for the adaptation. Two techniques are used for code generation: cascade adaptation mechanism and aspect-oriented adaptation enabling.

Cascade adaptation mechanism

We create two adaptive classes `AdaptivePipedInput` and `AdaptivePipedOutput` for the piped input and output, respectively. Figure 6.12 (a) shows the `read()` method for the `AdaptivePipedInput` class. The class comprises an object of the `sync.PipedInput` class (`syncInput`), and an object of the `async.PipedInput` class (`asyncInput`). The `read()` method of `AdaptivePipedInput` invokes the `read()` method of one of the objects based on its current state (lines 5 and 15). The invocation of `syncInput.read()` (line 8) (the source) is placed in a `try` block since we use the exception handling mechanism to implement the cascade adaptation. When an adaptation occurs during the invocation of the `read()` method, a state object of the type `SyncInputState` will be thrown as an exception (line 8), and caught by the `catch` block (lines 10–13), where the state object is stored in the variable `outgoingState`.

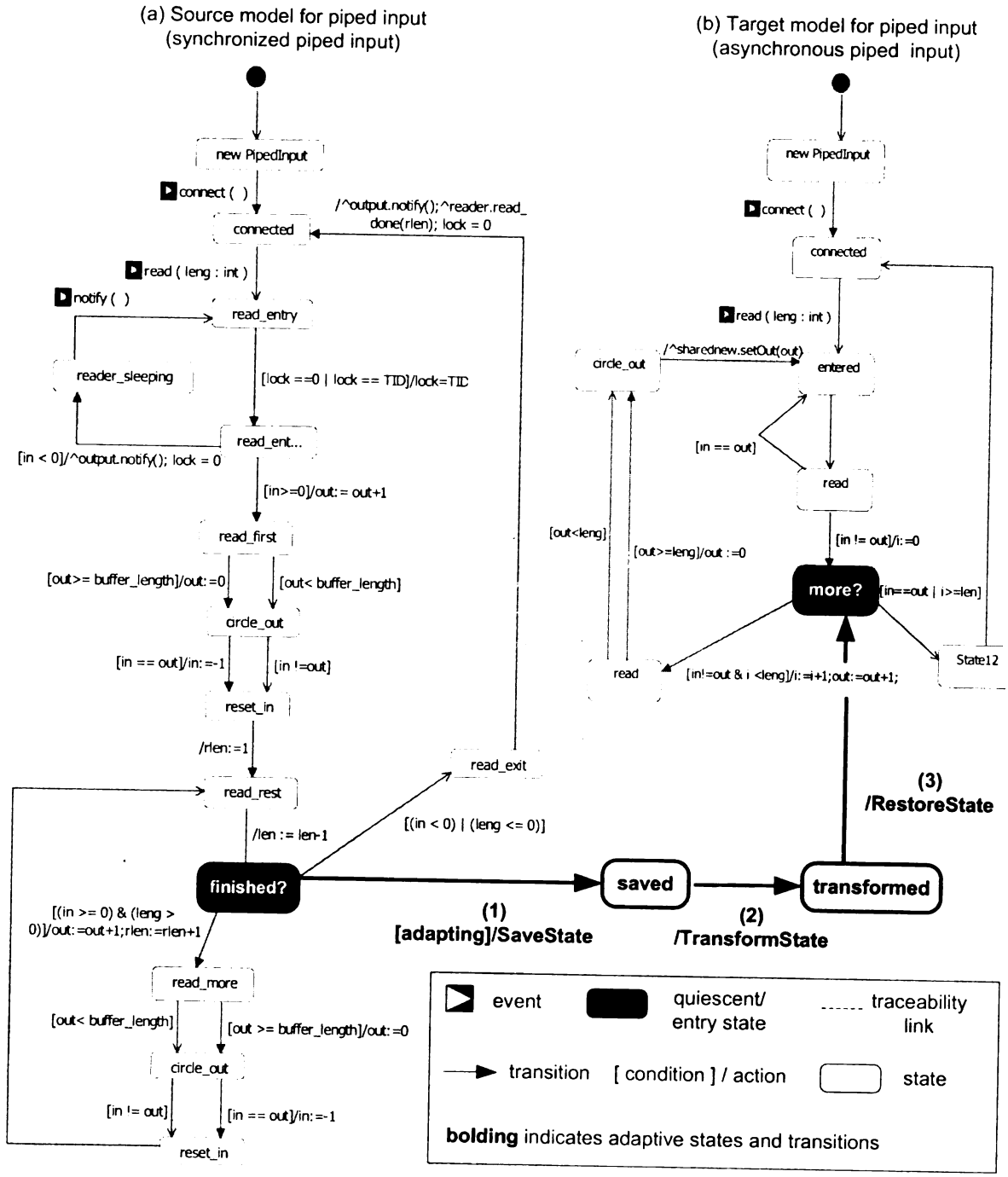


Figure 6.11: Adaptation model for the piped input class

At lines 20–21, a state object, `incomingStat`, for the asyncInput is created based on `outgoingState`, and at line 22, the `incomingStat` is passed to the `read()` method of `asyncInput` (the target) as an argument.

```
01 public synchronized int read(byte b[],
02                              int off, int len){
03   InputState outgoingState = null;
04   while(true){
05     if(!adapting && in_sync){
06
07       try{
08         int ret = syncInput.read(b, off, len);
09         return ret;
10       }catch(SyncInputState  stateObj){
11         outgoingState= stateObj;
12          adapting = true;
13       }
14
15     }else if(!adapting && in_async){
16       int ret = asyncInput.read(b,off,len);
17       return ret;
18
19     } else if(adapting){
20       AsyncInputState incomingStat=
21           new AsyncInputState(outgoingState);
22       int ret = asyncInput.read(incomingStat);
23       return ret;
24     }
25   }
26 }
```

(a) Java code for `AdaptivePipedInput.read()`

```
01 if(adapting()){
02   SyncInputState stateObj
03     = new InputState();
04   stateObj.in = in;
05   stateObj.len = len;
06   stateObj.off = off;
07   stateObj.rlen = rlen;
08   stateObj.b = b;
09   throw stateObj;
10 }
```

(b) Java code at quiescent state of `sync.PipedInput.read()`

```
01 public int read(
02     AsyncInputState stateObj){
03   byte b[] = stateObj.b;
04   int len = stateObj.len;
05   int rlen = stateObj.rlen;
06   int off = stateObj.off;
07   ......
08 }
```

(c) Java code at entry state of `async.PipedInput.read()`

Figure 6.12: The `read()` method for the adaptive piped input class

Since we have chosen the state finished? in the synchronized input model as the quiescent state, we follow the traceability links in Figure 6.10, and locate the point corresponding to the quiescent state (line 34 in Figure 6.10 (a)). We insert Java code in this line to implement the adaptation transitions.

The code inserted is shown in Figure 6.12 (b). If an adaptation request is present (line 1), then we create a state object `stateObj` (lines 2, 3) which saves the current state (lines 4–8). Then `stateObj` is thrown as an exception (line 9), which will be caught at line 10 of Figure 6.12 (a).

The entry point in the target is identified similarly. Figure 6.12 (c) shows the

Java code inserted at the entry point of the `read()` method of `async.PipedInput`. It first restores the program state from the input state object, `stateObj` (lines 3–6), then continues with other operations in the method thereafter.

We follow the same steps introduced above to generate the code for the `AdaptivePipedOutput` class as well.

Enable adaptation in legacy code

In the legacy Java program that uses the Java pipeline, we use an aspect-oriented approach to replace calls to the constructors of the non-adaptive piped input/output classes with those of the adaptive piped input/output classes. Figure 6.13 shows an aspect definition in AspectJ that enables adaptation in legacy Java programs. The aspect defines two pointcuts, `constructInput` (lines 5–7) and `constructOutput` (lines 9-11), to identify the invocations to the constructors of the piped input and output classes, respectively. The aspect also defines two advices, `PipedInput` (lines 13–15) and `PipedOutput` (lines 17-19), which respectively, replace the constructors of the non-adaptive classes with constructors of the adaptive classes. The aspect is then *woven* into the legacy program using the AspectJ compiler "ajc" [132].

After the above steps, the legacy Java program using the non-adaptive pipeline is transformed into a program using the adaptive pipeline, whose behavior has been verified against the set of requirements elicited in the requirements analysis.

6.4 Extensions

In this section, we discuss a number of extensions to the technique introduced in Section 6.2 to handle more general adaptation scenarios.

```
01 package Main;
02
03 public aspect Adaptenabling {
04
05   public pointcut constructInput():
06     call(sync.PipedInput.new(..)) ||
07     call(async.PipedInput.new(..));
08
09   public pointcut constructOutput():
10     call(sync.PipedOutput.new(..)) ||
11     call(async.PipedOutput.new(..));
12
13   PipedInput around():constructInput() {
14     return new AdaptivePipedInput();
15   }
16
17   PipedOutput around ():constructOutput() {
18     return new AdaptivePipedOutput();
19   }
20 }
```

Figure 6.13: AspectJ code for adaptation enabling

6.4.1 Collaborating Adaptive Components

We apply role-based design [135] to handle collaborations among adaptive components during adaptation. In Section 6.2, we restricted our discussion to independent adaptations of a single thread. In a distributed system, collaborating components in multiple threads are usually required to adapt in a coordinated fashion. Examples include the adaptation of an encoder filter and a decoder filter on the sender and receiver processes, respectively. We create an *adaptation collaboration Statechart diagram* (ACSD) with a number of predefined roles (e.g., encoder, decoder, etc) to model each such kind of collaboration. Each collaborating adaptive component registers with the ACSD as a role, and ACSD coordinates the adaptation of different roles in the collaboration by sending and receiving messages. The states in an ACSD represent different stages in the collaborative adaptation process; the transitions of an ACSD represent allowable adaptation sequences of the roles. The transitions are triggered by the receipt of messages from the roles. After both the adaptation models

and the ACSD are designed, the ACSD is verified along with the adaptation models. The ACSD is then translated into an *adaptation collaboration driver* class (centralized) with three primitive methods: (1) registering a role, (2) getting the current state of the adaptation, and (3) triggering an adaptive transition to the next state of the adaptation.

6.4.2 Adapting to Multiple Target Programs

In Section 6.2, we restricted our discussion to adaptations from a source program to only one target program. Our approach can be extended to support a more general case where a source program may adapt to different target programs under different triggering conditions. For each target program, we create a separate adaptation UML model and verify the models independently. The adaptation to multiple different target programs may not share the same quiescent states. To support multiple quiescent states in a source program, we create a different type of state object for each quiescent state. The corresponding state transformation routine is then determined based on the type of the state object and the target program.

6.5 Related Work

The work introduced in this chapter is directly related to two areas of research: using formal models for analysis and enabling adaptation in legacy code. The former has already been discussed in Section 5.5. Thus, we focus our discussion on the latter area.

Numerous techniques have been proposed to enable dynamic adaptation in legacy, non-adaptive software [47, 121, 139]. Sadjadi *et al* [121] introduced the *transparent shaping* (TRAP) technique to generate *adaptable programs* (i.e., programs whose behavior can be changed at run time) from existing application. Their ap-

proach comprises two steps. In the first step, occurring before run time (i.e., compile time or load time), they use static weaving techniques to weave *hooks* into the program. These hooks are *interceptors* inserted in the program that support run-time insertion and removal of adaptive code. In the second step, occurring at run-time, a *composer* manages the insertion and removal of adaptive code in the program using the hooks embedded in the first step. They have developed a Java implementation of the TRAP technique, TRAP/J [122], that uses the AspectJ compiler to enable dynamic adaptation in legacy Java programs. Both our approach and theirs are rooted from the aspect-oriented adaptation enabling technique introduced by Yang *et al* [139]. However, our focus is on assurance, while their focus is on the management of adaptation code, i.e., separating adaptation code from non-adaptive code. Their technique can be leveraged by ours to better manage the source code while providing assurance to adaptive software.

6.6 Discussion

In this chapter, 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 activities on three types of artifacts: the implementation in Java, the UML Statechart diagrams, and the formal models in Promela. The design of the adaptive programs is performed on the UML Statechart diagrams. The analysis for correctness is performed on the formal models. The final adaptive programs are generated for the implementation. The transformation of different types of artifacts is accomplished by using a combination of the metamodel-based technique, the cascade adaptation technique, and the aspect-oriented adaptation enabling technique.

We have made a few assumptions about a number of existing techniques that we leverage. These techniques must be more mature in order for the proposed approach to be practical in general. (1) Reverse engineering: In our approach, we require the extraction of a Statechart diagram from the legacy code. The metamodel-based technique introduced in this chapter handles only a small subset of all possible software. Reverse engineering for a general system is still an open research topic, including Bandera [28], which extracts finite-state models from Java source code. (2) Legacy code base: We require a legacy code base comprising a set of non-adaptive legacy programs and the properties associated with each program. The code base and the properties may not be available in some legacy software systems. (3) Model checking tools: Spin [59] originated in the telecommunications industry [60], but has gained increasing use in other industrial domains involving distributed systems, such as flight systems [42], railway systems [25], and systems that are fault-tolerant [123]. (4) Code generation tools: We briefly sketched a technique that translates UML adaptation models into Java code using the cascade adaptation mechanism. However, this technique does not handle recursions or exception handling. We have developed guidelines for the transformation between different types of artifacts using the metamodel-based technique. Thus far, the guidelines are intended to be systematically followed by developers, though they are amenable to automation. The technique is not yet automated, and therefore, the conformity between the models and the code still largely relies on the developers. The solutions to the above assumptions are much broader than the scope of this chapter or dissertation. Nonetheless, our approach provides a systematic approach to the development of adaptive software in which existing tools are applicable. This approach has been applied to a number of adaptive components for mobile computing applications that we have studied in our research, including the adaptive Java pipeline program and an adaptive forward error correction-based wireless communication program [16, 154]. We also hope that this chapter will help

motivate more research on realizing the above assumptions.

Part III

Implementation of Adaptive Software

Chapter 7

Modular Model Checking for Adaptive Software

In this chapter, we introduce a sound approach [145] for modularly verifying whether an adaptive program satisfies its requirements specified in LTL/A-LTL as a means to provide assurance to adaptive software in the implementation phase. Compared to existing adaptive program model checking techniques [2, 72], our approach has the following advantages: (1) It reduces the verification cost of adaptive programs by a factor of n, where n is the number of steady-state programs encompassed by the adaptive program. (2) Our verification technique can be applied incrementally to verify an adaptive system, leveraging previously verified results.

In Chapter 5 and Chapter 6, we have proposed using model checking to verify design models against global invariants and local properties specified in LTL for newly designed and re-engineered legacy adaptive software, respectively. Critical properties of an adaptive program need to be verified, including properties local to each steady-state program (local properties), properties that apply during adaptation from one steady-state program to another (transitional properties), and invariant properties that must be held by the adaptive program throughout its execution (global invari-

ants). Since the number n of steady-state programs in an n-plex adaptive program may be large, existing model checking approaches may be too expensive (in terms of time and space complexity) to verify large-scale adaptive programs. In addition, an adaptive program may be developed in a stepwise fashion as in extreme programming [56], where an $(n+1)^{\text{th}}$ steady-state program may be incrementally developed *after* an n-plex adaptive program has been developed and verified; the model checking approach should verify the adaptive program incrementally without repeating the model checking for the entire adaptive program.

This chapter proposes a modular model checking approach to address the above problems. We avoid directly model checking a large adaptive program by decomposing the task into smaller verification modules. First, we verify a set of *base conditions* (i.e., the properties that can be verified locally in each steady-state program) using a traditional model checking approach [51, 103, 136, 138]. Second, for each steady-state program P_i, we calculate the *guarantees*, the necessary conditions for P_i to satisfy its base conditions. Third, for each steady-state program P_j, we calculate the *assumptions*, the sufficient conditions for P_j to satisfy transitional properties and/or global invariants. Then we prove that all the assumptions and guarantees can be directly or indirectly inferred from the set of base conditions, thus completing the overall verification process.

Compared to existing approaches, our approach reduces the complexity of verifying an adaptive program by a factor of n, where n is the number of steady-state programs in the adaptive program. Moreover, when one steady-state program P_i is modified or a new steady-state program P_{n+1} becomes available after the remainder of the adaptive program has been verified using our approach, the model checking that needs to be repeated is limited to only P_i or P_{n+1} (and/or related adaptations). Also, our approach verifies not only LTL properties, but also A-LTL properties, whose verification, to the best of our knowledge, has not been published elsewhere. In this

chapter, we assume that the steady-state programs are given in the form of finite state machines (FSMs) introduced in Chapter 2.

We have proved the correctness of the proposed approach and implemented the algorithms in a prototype model checker AMOEBA (Adaptive program MOdular Analyzer) in C++. We have successfully applied AMOEBA to verify a number of adaptive programs developed for the RAPIDware project [95], including an adaptive TCP routing protocol [130] and the adaptive Java pipeline example [151]. The remainder of this chapter is organized as follows. In Section 7.1, we introduce the verification problems using an adaptive TCP routing protocol to illustrate the key verification challenges. Section 7.2 describes a number of basic algorithms and data structures and their properties that are used in our approach. Section 7.3 uses the TCP routing example to outline the basic idea of our approach. Section 7.4 formally describes the model checking algorithms and proves their soundness. Section 7.5 describes an example application of our technique. We discuss optimizations, complexity, and limitations of our approach in Section 7.6, and related work is overviewed in Section 7.7. Section 7.8 summarizes this chapter and briefly discusses possible extensions of our proposed approach.

7.1 Specifying Adaptive Systems

In this section, we introduce a simplified version of an *adaptive TCP routing protocol* [130] that provides a concrete demonstration for adaptive program verification challenges, and it is also used to illustrate our proposed solution.

7.1.1 Adaptive TCP Routing

The adaptive TCP routing protocol is a network protocol involving adaptive middleware in which a router balances the network traffic by dynamically choosing

the next hop for delivering packets. We consider two types of next hop nodes: *trusted* and *untrusted*. The protocol works in two different modes: safe and normal. In the safe mode, only trusted nodes are selected for packet delivery, and in the normal mode, both types are used in order to maximize throughput. Any packet must be encrypted before being transferred to an untrusted node. We consider the program running in the safe and normal modes to be two *steady-state programs* P_1 and P_2, respectively. Figure 7.1 shows the FSM for the adaptive protocol program. For convenience, we assign a unique *state name* to each state. The upper rectangle in Figure 7.1 illustrates P_1. Initially, P_1 is in the ready1 state, in which, P_1 may receive a packet and move to state received1. At this point, P_1 searches for a trusted next hop and moves to state routed1. Then P_1 sends the packet to the next hop and goes to state sent1, and returns to the ready1 state. The lower rectangle in Figure 7.1 illustrates P_2. The ready2, received2, and sent2 states are similar to those in P_1. In state received2, searching for the next hop may return a trusted or untrusted node, and thus P_2 moves to the safe2 and unsafe2 states, respectively. From state unsafe2, P_2 tests whether the input packet has been encrypted. If so, then P_2 goes to state encrypted2, otherwise, it first goes to state unencrypted2 and then encrypts the packet and goes to the encrypted2 state. Four adaptive transitions are defined between P_1 and P_2: a1, a2, a3, and a4. We annotate (in italics) each state with the conditions that are true for the state (only the relevant conditions are shown).

Critical properties of the adaptive program are specified as global invariants LTL [111]. For the adaptive routing protocol, we require the program not to drop any packet throughout its execution, i.e., after it receives a packet, it should not receive the next packet before it sends the current packet. Formally in LTL:

$$inv = \Box(received \Rightarrow (\neg ready\, \mathcal{U}\, sent)). \tag{7.1}$$

In addition to global invariants, we also specify local properties for each steady-state

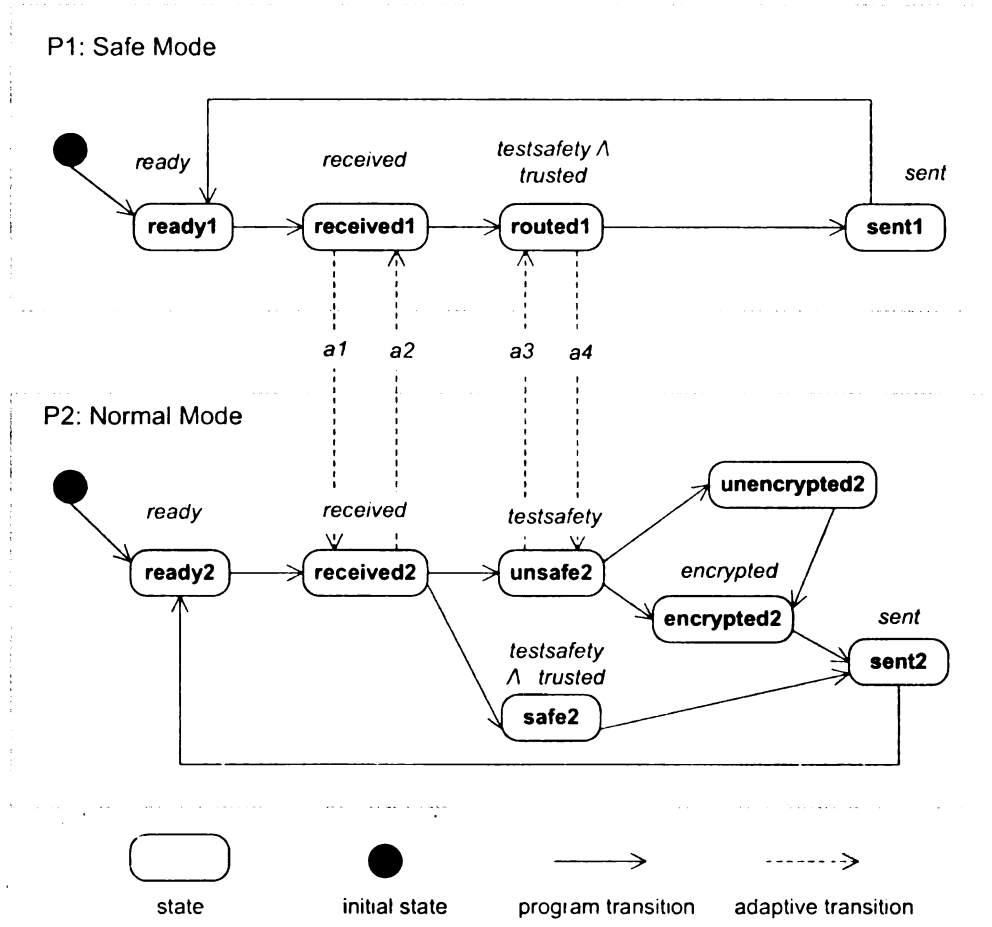


Figure 7.1: Case study: adaptive routing protocol

program. In this example, we require P_1 to never use an untrusted (unsafe) next hop. Formally in LTL, we write

$$LP_1 = \Box(\neg unsafe), \tag{7.2}$$

where $unsafe \equiv testsafety \Rightarrow !trusted$, which is true only in state unsafe2. For P_2, we require the system to encrypt a packet before sending the packet if the next hop is unsafe. Formally in LTL, we write

$$LP_2 = \Box(unsafe \Rightarrow (\neg sent\, \mathcal{U}\, encrypted)). \tag{7.3}$$

For an execution of an adaptive program, if it adapts among the steady-state

programs of the adaptive program, then the execution must satisfy the corresponding local properties of the steady-state programs sequentially in the same order. We discussed this type of properties (transitional properties) in Section 3.4.2 and proposed using A-LTL and sequential composition to specify transitional properties. The kind of transitional properties we discuss in this chapter are restricted to those based on one-point adaptations semantics (see Section 3.3.1). That is, we verify that in an adaptive program with n steady-state programs $P_1, P_2, \cdots, P_n$, any execution σ_j with a sequence of $(k-1)$ steps of adaptations, starting from P_{j_1}, going through $P_{j_2}, \cdots, P_{j_k}$ satisfies

$$LP_{j_1} \overset{\Omega_{j_1}}{\rightharpoonup} LP_{j_2} \overset{\Omega_{j_2}}{\rightharpoonup} LP_{j_3} \cdots \overset{\Omega_{j_{k-1}}}{\rightharpoonup} LP_{j_k}, \text{ where } j_i \neq j_{i+1}.$$

For simplicity, in the examples in this chapter we assume $\Omega \equiv \mathit{true}$ and write $\phi \rightharpoonup \psi$, instead of $\phi \overset{\Omega}{\rightharpoonup} \psi$. However, our approach also applies to cases where Ω can be an arbitrary LTL formula.

In the adaptive routing protocol, we express the transitional property that must be satisfied by executions adapting from P_1 to P_2 with the A-LTL formula $LP_1 \rightharpoonup LP_2$, and the transitional property that must be satisfied by executions adapting from P_1 to P_2 and then to P_1 with the A-LTL formula $LP_1 \rightharpoonup LP_2 \rightharpoonup LP_1$, etc. Note that each different adaptation sequence corresponds to a different transitional property. Since in a general adaptive program, there are infinite numbers of different possible adaptation sequences, the number of possible transitional properties is also infinite.

7.1.2 Verification Challenges

Regarding an adaptive program, we must achieve two verification goals: (1) the global invariants hold for the adaptive program regardless of adaptations, and (2) when the program adapts within its steady-state programs, the corresponding transitional properties are satisfied.

Model checking techniques determine whether a program satisfies a given temporal logic formula by exploring the state space of the program. Numerous model checking techniques have been proposed to verify various properties of different types of programs. However, to the best of our knowledge, none of the existing verification approaches can be applied to verify adaptive programs efficiently (in terms of time and space complexity), which is discussed next.

Global invariants. Allen *et al* [2] used model checking to verify that an adaptive program adapting between two steady-state programs satisfies certain global properties. While they do not explicitly address adaptations of n-plex adaptive programs (for $n > 2$), a straightforward extension could be to apply *pairwise model checking* between each pair of steady-state programs, separately. The first drawback of the pairwise extension is that it requires n^2 iterations of model checking for a program with n steady-state programs. More importantly, this extension is theoretically unsound since it verifies executions with only one adaptation step, and thus it does not guarantee the correctness of executions with more than one adaptation step.

A sound solution proposed by Magee [93], called the *monolithic* approach [27], treats an adaptive program as a general program and directly verifies the adaptive program against its global invariants. The monolithic approach suffers from the state explosion problem: It is well known that the major limiting factor in model checking is the large amount of memory required by the computation [29]. However, most known efficient model checking algorithms have space complexity $O(nlogn)$ [29], where n is the size (i.e., the number of states and transitions) of the program under verification. Although the example adaptive routing protocol has only two steady-state programs, the number of steady-state programs in an adaptive program can easily become very large in practice, and so can the required memory by the model checking computation. Furthermore, the monolithic approach is not suited for incremental adaptive software

development. For example, after the adaptive routing protocol with two steady-state programs is verified, if a third steady-state program for a different condition becomes available by incremental development, then the monolithic approach cannot leverage the existing verification results. Instead, the entire verification must be repeated for the adaptive program with three steady-state programs.

Transitional Properties. The transitional property verification is even more challenging. Since executions of an adaptive program may adapt within its set of steady-state programs in an infinite number of different sequences, the number of different transitional properties is also infinite. Therefore, it is impossible to verify each transitional property separately. To the best of our knowledge, no existing approaches address the transitional property verification problem defined in this chapter.

In order to address the above problems, we propose a sound modular model checking approach for adaptive programs against their global invariants and transitional properties that not only reduces verification complexity by a factor of n, where n is the number of steady-state programs, but also reduces verification cost by supporting verification of incrementally developed adaptive software.

7.2 Preliminary Algorithms and Data Structures

This section introduces preliminary notations, algorithms, and basic data structures that are required by our model checking algorithm. We define an *obligation* of a state s of a program P to be a necessary condition that the state must satisfy in order for the program to satisfy a given temporal logic formula ρ. We describe the *Partitioned Normal Form* (PNF) that is used to propagate the obligations for analysis. Then we overview the property automaton needed to support the processing of the obligations. Next, we introduce an algorithm that marks each state of a program

with a set of obligations. Intuitively, the algorithm first marks the initial states of P with obligation ρ, then the obligations of each state are propagated to its successor state(s) in a way that preserves the necessary conditions along the propagation paths. If a state is reachable from the initial states from multiple paths, then the obligations of the state is the conjunction of the necessary conditions propagated to the state along all these paths.

7.2.1 Partitioned Normal Form

The logic closest to A-LTL is the ITL studied by Bowman and Thompson [17]. They have introduced using the *Partitioned Normal Form* (PNF) [17] to support model checking ITL properties. In our work, we also use PNF to handle obligation propagation. We rewrite each A-LTL/LTL formula into its PNF as follows:

$$(pe \wedge empty) \quad \vee \quad \bigvee_i (p_i \wedge \bigcirc q_i). \tag{7.4}$$

The expression, $(pe \wedge empty)$, depicts the condition when a sequence is *empty*, i.e., the last state of the sequence, where pe is a proposition that must be satisfied by the last state. The expression, $\bigvee_i (p_i \wedge \bigcirc q_i)$, depicts the condition when the sequence is not the last state. The propositions p_i partition *true*, and q_i is the corresponding condition that must hold when p_i holds in the current state. Formally, pe, p_i, and q_i satisfy the following constraints:

- pe and p_i are all propositional formulae.
- p_i partitions *true*, i.e., $\bigvee_i p_i \equiv true$ and $p_i \wedge p_j \equiv false$ for all $i \neq j$.

All A-LTL/LTL formulae can be rewritten in PNF by applying PNF-preserving rewrite-rules [17]. For any A-LTL formulae ϕ, ψ, and Ω, the rewrite-rules are defined as follows.

The negation rule:

$$\neg\phi \;=\; (\neg p_e^{\phi} \wedge empty) \vee \bigvee (p_i^{\phi} \wedge \bigcirc \neg q_i^{\phi}). \tag{7.5}$$

The conjunction rule:

$$\phi \wedge \psi \;=\; (empty \wedge (p_e^{\phi} \wedge p_e^{\psi})) \vee \bigvee_i \bigvee_j (p_i^{\phi} \wedge p_j^{\psi}) \wedge \bigcirc (q_i^{\phi} \wedge q_j^{\psi}). \tag{7.6}$$

The disjunction rule:

$$\phi \vee \psi \;=\; (empty \wedge (p_e^{\phi} \vee p_e^{\psi})) \vee \bigvee_i \bigvee_j ((p_i^{\phi} \wedge p_j^{\psi}) \wedge \bigcirc (q_i^{\phi} \vee q_j^{\psi})). \tag{7.7}$$

The next rule:

$$\bigcirc\phi \;=\; (empty \wedge false) \vee (true \wedge \bigcirc\phi). \tag{7.8}$$

The global rule:

$$\Box\phi \;=\; (empty \wedge p_e) \vee \bigvee (p_i \wedge \bigcirc (q_i \wedge \Box\phi)). \tag{7.9}$$

The eventuality rule:

$$\Diamond\phi \;=\; (empty \wedge p_e) \vee \bigvee (p_i \wedge \bigcirc (q_i \vee \Diamond\phi)). \tag{7.10}$$

The until rule:

$$\phi \mathcal{U} \psi \;=\; (empty \wedge p_e^{\psi}) \vee \bigvee_i \bigvee_j (p_i^{\psi} \wedge p_j^{\phi} \wedge \bigcirc(q_i^{\psi} \vee q_i^{\phi} \wedge \phi \mathcal{U} \psi)). \tag{7.11}$$

The adapt rule:

$$\begin{aligned} \phi \overset{\Omega}{\rightharpoonup} \psi \;=\; & (empty \wedge false) \vee \\ & \bigvee_i \bigvee_j (p_i^{\phi} \wedge p_j^{\Omega} \wedge pe^{\phi} \wedge \bigcirc((q_j^{\Omega} \wedge \psi) \vee (q_i^{\phi} \rightharpoonup \psi))) \vee \\ & \bigvee_i (p_i^{\phi} \wedge \neg pe^{\phi} \wedge \bigcirc(q_i^{\phi} \rightharpoonup \psi)). \end{aligned} \tag{7.12}$$

We use superscripts on p_i, q_i, and pe to represent the formula from which they are constructed. Since pe and p_i are all propositions, their truth values can be directly evaluated over the label of each single state. Therefore, the obligations of a given state can be expressed solely by a next state formula, which will be the q_i part of a disjunct if the state has successor states, or *empty* if the state is a deadlock state.

7.2.2 Property Automaton

Bowman and Thompson's [17] tableau construction algorithm first creates a *property automaton* based on an initial formula ϕ, and then constructs the *product automaton* of the property automaton and the program. Their approach is suited for verifying that all initial states satisfy the same initial formula.

However, our model checking algorithm requires us to mark program states with necessary/sufficient conditions for different initial states to satisfy different initial formulae in the assumption computation step. We could create a different property automaton for each formula, but it would have duplicate states. Instead, we extend

their property automaton construction algorithm to support multiple initial formulae for our purpose as follows.

A property automaton is a tuple (S, S_0, T, P, N), where

- S is a set of states.
- S_0 is a set of initial states where $S_0 \subseteq S$.
- $T : S \leftrightarrow S$ maps each state to a set of next states.
- $P : S \rightarrow proposition$ represent the propositional condition that must be satisfied by each state.
- $N : S \rightarrow formula$ represents the condition that must be satisfied by all the next states of a given state.

Property automaton construction algorithm: Given a set of A-LTL/LTL formulae Φ, we generate a *property automaton* $PROP(\Phi)$ with the following features:

- For each member $\phi \in \Phi$, create an initial state $s \in S_0$ such that $P(s) = true$, $N(s) = \phi$.
- For each state $s \in S$, let the PNF of $N(s)$ be $(pe \wedge empty) \vee \bigvee_i (p_i \wedge \bigcirc q_i)$, then
  - if the state $s'_i = (p_i, q_i)$ does not exist in S, create s'_i and add s'_i to S;
  - make s'_i a successor state of s.

A *path* of a property automaton is an infinite sequence of states $s_0, s_1, \cdots$ such that $s_0 \in S_0$, $s_n \in S$, and $(s_i, s_{i+1}) \in T$, for all i $(0 \leq i < n)$. We say a path of a property automaton $s_0, s_1, \cdots$, *simulates* an execution path of a program $s'_1, s'_2, \cdots$,

if $P(s_i)$ *agrees with* s'_i for all i $(i > 0)$.[1] We say a property automaton *accepts* an execution path from an initial state $s \in S_0$, if there is a path in the property automaton starting from s that simulates the execution path. The property automaton constructed above, from initial state $s \in S_0$, accepts exactly the set of executions that satisfy $N(s)$.[2]

7.2.3 Product Automaton Construction and Marking

Our algorithm handles the case when each initial state of a program P is required to satisfy a different A-LTL/LTL formula. Given a program $P = (S^P, S_0^P, T^P, L^P)$ and a formula mapping function $\Psi : S_0^P \rightarrow$ A-LTL/LTL, we use the following algorithm to mark the states of P with sufficient/necessary conditions in order for P to satisfy Ψ. We can prove that the algorithm works for calculating both assumptions and guarantees.

A *product automaton* Q is defined to be a tuple (S^Q, S_0^Q, T^Q), where

- S^Q is a set of states with two fields: $(pstate, nextform)$. The *pstate* field represents a state of program P. The *nextform* field contains an A-LTL/LTL formula declaring what should be true in the next state.
- S_0^Q is a set of initial states, where $S_0^Q \subseteq S^Q$.
- T^Q is a set of transitions, where $T^Q : S^Q \times S^Q$.

Product automaton construction algorithm: Given a program automaton P, and a mapping function Ψ from its initial states to a set of A-LTL/LTL formulae, we generate a product automaton $PROD(P, \Psi)$ as follows:

1. Calculate the relational image Φ of the initial states of P under the mapping function Ψ.

[1] p agrees with q, if and only if $p \wedge q \not\equiv false$.

[2] We ignore the eventuality constraint [103] (a.k.a. self-fulfillment [88]) at this point. However, later steps will ensure eventuality to hold in our approach.

$$\Phi = \{\phi \mid \exists\, s \in S_0,\ \phi = \Psi(s)\}.$$

2. Construct a property automaton $PROP(\Phi)$ with the set of initial formulae Φ using the property automaton construction algorithm introduced in Section 7.2.2.

3. For each initial state of the program $s_i \in S_0^P$, add a pseudo program state ps_i (a program state that was not originally in the state space S^P) and a transition (ps_i, s_i) to P, and label ps_i with $L^P(ps_i) = true$.

4. For each pseudo program state ps_i, create an initial product state s_0^Q, where $s_0^Q \in S_0^Q$ and $s_0^Q = (ps_i, \phi)$.

5. Create the cross product of P and $PROP(\Phi)$ from the above initial states [17, 103].

State marking algorithm: Given a program P and an initial formula mapping function Ψ, we first construct the product automaton $PROD(P, \Psi)$, then we construct the *marking of each program state* $MARK(s)$ to be the set of *nextform* fields of states in $PROD(P, \Psi)$ that correspond to s. Our *marking algorithm* generates the function $MARK$ over selected states. We also apply optimizations to the algorithm so that we do not need to store the entire product automaton.

Theorem 6: *The marking algorithm will terminate.*

Proof 6: *The set of markings of each state is a subset of the formulae in the property automaton, i.e., the number of formulae in each marking is finite. Also the number of states in the property automaton is finite. We repeatedly check all nodes to see whether there will be update to the node. For each iteration, we have two possible outcomes:*

1. *If no change is made to the markings, then the process terminates.*

2. *If at least one formula is added to the marking of a state, then we will repeat the process.*

Since the number of nodes and the number of markings in each node are both finite, and the total number of formulae in all the state markings strictly increases by at least 1 *in each update, the process has only a finite number of iterations before it eventually terminates.*

The markings generated by the marking algorithm contribute to the assumptions and guarantees in our model checking approach. We prove that the marking of a state contains the necessary conditions that the state must satisfy in order for the program to satisfy Ψ.

Theorem 7: *For a program P with initial states S_0 and an initial formula mapping function Ψ, let MARK be the marking for the states generated using the marking algorithm above, and let θ be the conjunction of the marking of a state s, i.e., $\theta = \bigwedge MARK(s)$, then P satisfies Ψ implies that s satisfies θ. That is,*

$$(P \models \Psi) \Rightarrow (s \models \bigwedge MARK(s)). \tag{7.13}$$

Lemma 1: *A path π of a property automaton A simulates an execution σ, then $\sigma \models N(\pi_0)$ if and only if $\sigma^1 \models N(\pi_1)$. (Note that π may or may not be a self-fulfilling path.[3])*

This is a direct result of building the property automaton according to the PNF for a formula. Since π simulates σ, we establish that π_1 agrees with σ_0. Let $N(\pi_0)$ be $(pe \wedge empty) \vee \bigvee_i (p_i \wedge \bigcirc q_i)$. Without losing generality, we assume $P(\pi_1) = p_1$, and

[3]A path is self-fulfilling [88], if and only if it reaches accepting states infinitely often.

$N(\pi_1) = q_1$. We have $\sigma \models N(\pi_0) \Leftrightarrow \sigma \models \bigcirc q_1$. Therefore, $\sigma \models N(\pi_0)$ if and only if $\sigma^1 \models N(\pi_1)$.

Lemma 2: *A path π of a property automaton A simulates an execution σ, then*

$$\sigma \models N(\pi_0) \Leftrightarrow \sigma^i \models N(\pi_i). \tag{7.14}$$

This is an inductive result of Lemma 1.

Now we prove Theorem 7.

Proof 7:

Prove by contradiction.

Assume that s does not satisfy θ. There must exist a path σ from s such that $\sigma \not\models \theta$. Since $\theta = \bigwedge MARK(s)$, there must exist a formula $\phi \in MARK(s)$ such that $\sigma \not\models \phi$. $\phi \in MARK(s)$ implies that (s, ϕ) is reachable by a path π in the product automaton starting from some $s_0 \in S_0$ and $\Psi(s_0)$. Let σ' be the finite execution corresponding to π. From Lemma 2, we have $\sigma' \frown \sigma \models \Psi(s_0) \Leftrightarrow \sigma \models \phi$. Given that the program satisfies Ψ, we have $\sigma \models \phi$: a contradiction.

In fact, we can also prove that for a state s, if s satisfies all the formulae in the marking of s, then all paths starting from $s_i \in S_0$ going through s satisfy $\Psi(s_i)$ [145].

Theorem 8: *For a program P with initial states S_0 and an initial formula mapping function Ψ, using the marking procedure above, for any state s of P, let θ be the conjunction of all the marking of s: $\theta = \bigwedge MARK(s)$, then s satisfies θ implies all paths of P starting from $s_i \in S_0$ going through s satisfy $\Psi(s_i)$.*

Proof 8:

Prove by contradiction.

Assume $s \models \theta$ *and there exists a path* σ *of* P *such that* σ *starts from* $s_i \in S_0$ *and goes through* s, *and* $\sigma \not\models \Psi(s_i)$.

Let s *be the* j^{th} *state in* σ, *and let* π *be the path in the product automaton corresponding to* σ. $N(\pi_j)$ *must be a conjunct in* θ. *Therefore, we have* $s \models N(\pi_j)$, *and thus* $\sigma^j \models N(\pi_j)$.

From Lemma 2, we have $\sigma \models N(\pi_0) \Leftrightarrow \sigma^j \models N(\pi_j)$. *Then we have* $\sigma \models N(\pi_0)$, *which implies* $\sigma \models \Psi(s_i)$: *a contradiction.*

Now, we introduce a property of the markings that allows us to compose and reason about already verified steady-state programs without repeating the verification.

Theorem 9: *For a program* P_i *with initial states* S_0 *and an initial formula mapping function* Ψ, *using the marking algorithm above, for any state* s *of* P_i, *let* θ *be the conjunction of all the markings of* s: $\theta = \bigwedge MARK(s)$. *Let* P_j *be a state machine. Let* P *be the state machine including* P_i, P_j *and transitions connecting a state* s_i *of* P_i *with some states in* P_j. *Let* $P_{i,j}$ *be the state machine constructed from* s_i, P_j, *and the transitions between* s_i *and* P_j. *The following two statements are equivalent:*

- *All states in* P_i *still satisfy their markings in* P.
- s_i *satisfies all its markings in* $P_{i,j}$.

The theorem is illustrated in Figure 7.2.

Proof 9:

- *We prove that if all states in* P_i *still satisfy all their markings in* P, *then* s_i

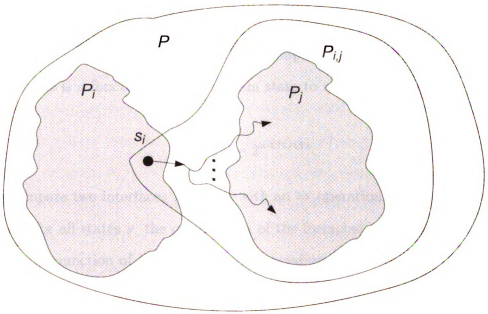


Figure 7.2: Illustration for Theorem 9

satisfies its marking in $P_{i,j}$. Since s_i is one of those states, and $P_{i,j}$ is a subset of P, the conclusion is a direct result of the condition.

- *We prove that if s_i satisfies all its markings in $P_{i,j}$, then all states of P_i satisfy their markings in P.*

  Let c be an arbitrary condition in the marking of an arbitrary state s_k of P_i. For any self-fulfilling path σ starting from s_k, one of the following must be true:

  - *The path is not adaptive: Since this path does not include any adaptive transition from s_i, it is also a path of P_i. Therefore, $\sigma \models c$ (Theorem 7).*
  - *The path goes through an adaptive transition through s_i: Let σ_t be the last occurrence of s_i in σ. The condition c' propagated from c in s_k to s_i along the path $\sigma_0, \sigma_1, \cdots, \sigma_t$ must be in the marking for s_i and we have $\sigma^t \models c'$, where σ^t is the t^{th} suffix of σ. Thus, we have $\sigma \models c$.*

  Therefore, we have $\sigma \models c$ in either case.

7.2.4 Interface Definition

We use an *interface* structure to record assumptions and guarantees. An interface I of a program is a function from a program state to a set of A-LTL/LTL formulae.

$$I : S(P) \rightarrow 2^{\text{A-LTL/LTL}}. \tag{7.15}$$

We can compare two interfaces I_1 and I_2 with an $\Rightarrow$ operation, which returns *true* if and only if for all states s, the conjunction of the formulae corresponding to s in I_1 implies the conjunction of the formulae corresponding to s in I_2.

$$I_1 \Rightarrow I_2 \equiv \forall\, s : S(P), I_1(s) \Rightarrow I_2(s). \tag{7.16}$$

We also define a special *top* interface $\top$, which will serve as the initial value in the model checking algorithms presented in the next section. We use a simple λ-calculus [8] notation to define functions, where $\lambda(x : X).f(x)$ denotes an anonymous function that maps each element in domain X to its value $f(x)$.

$$\top \equiv \lambda(x : S(P)).true. \tag{7.17}$$

The formula states that $\top$ is a function that maps all states to *true*.

7.3 Modular Verification

This section introduces a modular approach to the model checking of adaptive programs, where the verification result of an adaptive program can be derived from the model checking results of its individual steady-state programs. Our modular model checking uses the *assume/guarantee* reasoning [27, 63, 64, 74, 101], where for a given program module (i.e., a steady-state program), the assumptions are the conditions of the running environment of the module that are assumed to be true, and guarantees

are the assurances provided by the module under the assumptions.

We first verify a set of base conditions in each steady-state program locally by using a traditional model checking approach [51, 103, 136, 138]. Second, for each steady-state program P_i we calculate the guarantees, the necessary conditions for each steady-state program to satisfy its base conditions locally. Third, for each steady-state program P_j, we calculate the assumptions, the sufficient conditions that each state of P_j must satisfy in order for P_j to satisfy transitional properties and/or global invariants. Fourth, we determine whether the guarantees logically imply the assumptions. If so, we conclude that all the assumptions and guarantees can be directly or indirectly inferred from the set of base conditions, thus completing the verification. Otherwise, a counterexample is generated.

Next, we describe several preliminary algorithms and data structures used in our approach, and then illustrate our proposed approach using the adaptive routing protocol by verifying its global invariants and then its transitional properties.

7.3.1 Global Invariant Verification

The global invariant verification proceeds as follows:

1. **Verify base conditions**: Verify each steady-state program against the global invariants individually.
2. **Compute guarantees**: Mark each state of the steady-state programs with conditions that are satisfied by those states when there is no adaptation.
3. **Compute assumptions**: Mark each state of the steady-state programs with conditions that must be satisfied by the state in order for the adaptive program to satisfy the global invariants.
4. **Compare guarantees with assumptions**: If the guarantees imply the assumptions, then the process returns success. Otherwise, it returns with a counterexam-

ple.

Next, we explain in further detail and illustrate each of these steps.

(1) Verify base conditions.

In this step, we verify each steady-state program against the global invariants individually. Since the global invariants are specified in LTL and we assume each steady-state program P_i to be non-adaptive itself, we use an existing LTL model checking algorithm provided in Spin [61] to verify the base conditions. By model checking, we determine that both P_i and P_j individually satisfy the invariant *inv*.

(2) Compute guarantees.

Next, we use a *marking algorithm* introduced in Section 7.2.3 to mark each state of the steady-state programs with *obligations*, i.e., conditions satisfied by the state when there is no adaptation. The obligations are true in each state if the steady-state program comprising the state has passed the base condition verification in step (1).

According to the atomic propositions shown in Figure 7.1, since P_1 satisfies *inv*, we conclude that state **ready1** satisfies *inv*, therefore, we mark **ready1** with the obligation *inv*. Then we propagate this obligation to its successor state **received1** as follows: First, it satisfies *inv*. Second, since *received* is *true* in the state, it must also satisfy $\neg ready\, \mathcal{U}\, sent$. Therefore, we mark **received1** with obligations *inv* and $\neg ready\, \mathcal{U}\, sent$. Similarly, the markings of states are repeatedly propagated to their successors. If an obligation is propagated to an already marked state through a different execution path, and the obligation is not in the existing marking of the state, then the state marking will be updated with the conjunction of the existing marking and the new obligation, otherwise the new obligation is ignored. This process will eventually converge (Theorem 6) and at that point (*fixpoint*), no state markings will be updated any further. Figure 7.3 shows the result of applying the marking

algorithm to P_1, where the guarantee markings are prefixed with "$g.*$". (The "$a.*$" denotes assumptions described in the next step.) Similarly, we apply the marking algorithm to P_2; results are shown in the bottom portion of Figure 7.3.

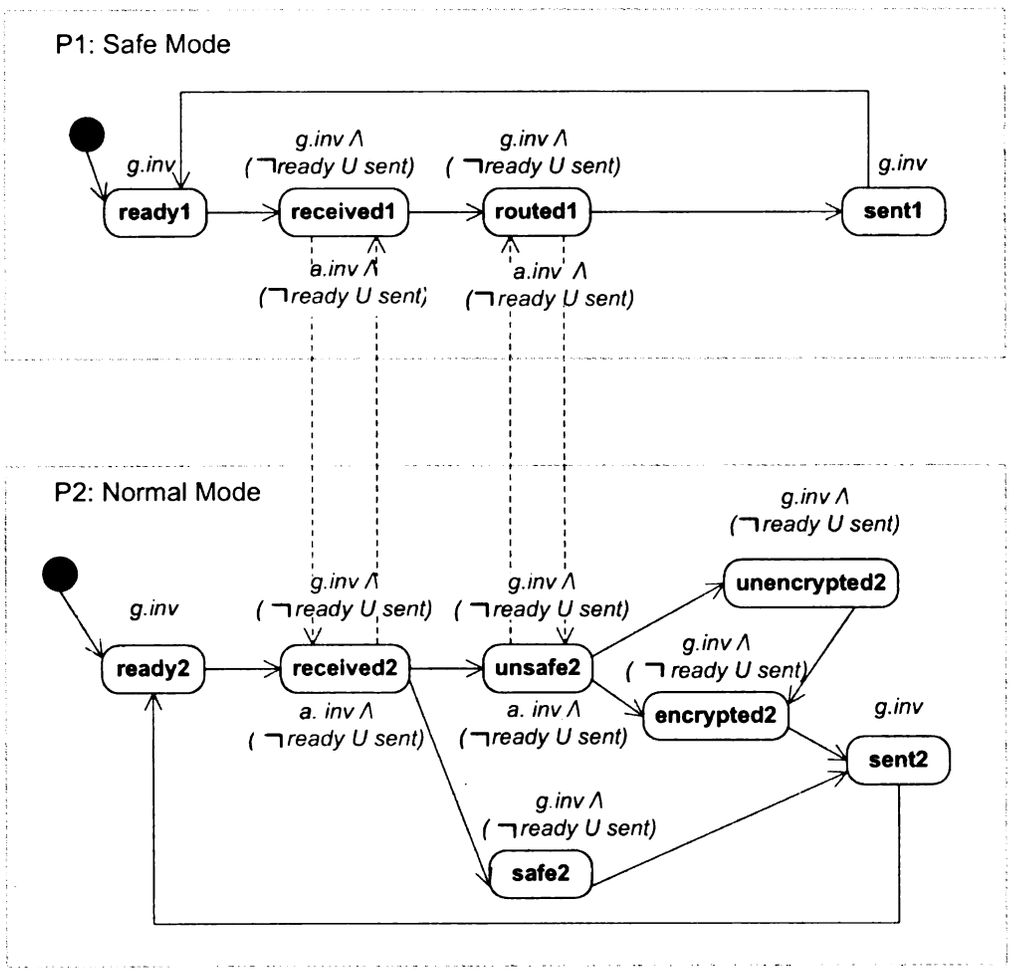


Figure 7.3: Markings for global invariant *inv*

In our algorithm, the obligations are propagated in such a way that guarantees each state satisfies all the obligations in its markings when there is no adaptation. We call these markings *guarantee markings*. We have proved in Theorem 9 that at the fixpoint, the markings of a steady-state program P_i have the following property, which is a **key insight** of this approach:

When a state machine P_j is connected to a state s in P_i with transi-

tions from s to states in P_j, all the states of P_i still satisfy their guarantees in the new program if and only if s still satisfies its guarantee in the new program.

(3) Compute assumptions.

Starting from the guarantee markings generated in step (2), for each state in each steady-state program P_i (the source) with outgoing adaptive transitions, we propagate the obligations along the adaptive transitions until reaching states of a target steady-state program P_j. Then we mark the reached states in P_j with the propagated obligations, which we call the *assumption markings* (denoted by prefix "$a.*$" in Figure 7.3).

In the adaptive routing protocol, we propagate the markings of received1 to received2 along the adaptive transition a1 and mark received2 with *inv* and $\neg ready\,\mathcal{U}\,sent$. Our process ensures that the assumption marking includes the **exact** set of conditions that received2 must satisfy in order for all executions starting from ready1, taking adaptive transition a1, and taking no further adaptations, to satisfy the global invariant *inv*. Similarly, we propagate the marking of routed1 to unsafe2, from received2 to received1, and from unsafe2 to routed1, respectively.

(4) Compare guarantees with assumptions.

Next we compare the guarantees with the assumptions. For each state, if the conjunction of its guarantee marking logically implies the conjunction of its assumption marking (checked automatically), then the process returns success, otherwise, it returns with a counterexample. For example, the guarantee marking for received2 indeed implies the assumption marking for received2. This result implies that all executions starting from ready1, taking adaptive transition a1, with no adaptation

thereafter, satisfy *inv*. We perform the comparison on every state of the steady-state programs with incoming adaptive transitions. Successful comparisons guarantee that any execution starting from ready1 or ready2, undergoing one step of adaptation, satisfies *inv*. If the guarantee of a state does not imply its assumption, then we generate a counterexample showing the path violating the global invariant. The counterexample feature is illustrated in the discussion for the transitional properties below.

7.3.2 Transitional Properties

The steps for transitional properties verification are similar to those used for global invariants. The first two steps are the same as the first two steps used for global invariant verification except that the LTL formulae that we verify are the local properties LP_1 and LP_2 instead of *inv*. The last two steps are described in detail below.

(1') Verify base conditions.

This step is the same as step (1) for global invariants except that we model check the local properties instead.

(2') Compute guarantees.

This step is the same as step (2) for global invariants except that we compute guarantees by marking the initial states with local properties. The guarantee markings after applying steps (1') and (2') are shown in Figure 7.4, prefixed with "$g.*$".

(3') Compute assumptions.

In this step, we start from the guarantee markings generated in the previous step. For each state in a steady-state program P_i with outgoing adaptive transitions going towards program P_j, we generate an obligation $\phi \rightarrow LP_j$ from each condition

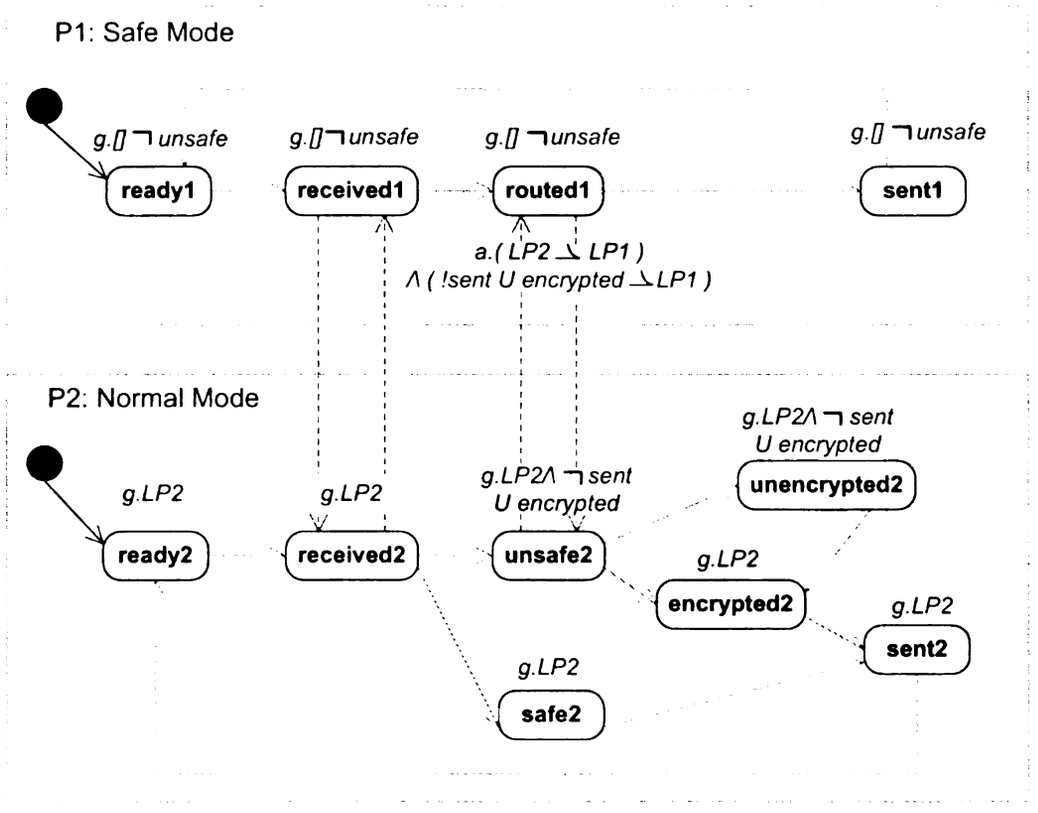


Figure 7.4: Markings for transitional properties

ϕ in its guarantee marking, where LP_j is the local property for P_j. Then we propagate the generated obligations to the states in P_j along the adaptive transitions to form their assumption markings. For example, the guarantee marking for **unsafe2** is LP_2 and $(\neg sent\,\mathcal{U}\,encrypted)$. From this marking, we generate obligations $LP_2 \rightarrowtail LP_1$ and $(\neg sent\,\mathcal{U}\,encrypted) \rightarrowtail LP_1$, respectively. These obligations are propagated to the state **routed1**, then we generate the assumption marking $LP_2 \rightarrowtail LP_1$ and $(\neg sent\,\mathcal{U}\,encrypted) \rightarrowtail LP_1$ for **routed1**. We repeat this process for all states in P_1 and P_2 with outgoing adaptive transitions, and the resulting assumption markings are shown in Figure 7.4, prefixed with "$a.*$".

(4') Compare guarantees with assumptions.

We compare the assumption markings with the guarantee markings of all states to see whether the assumptions are implied by the guarantees. If so, then the model

checking returns success, otherwise, it returns with a counterexample. We will prove in Section 7.4 that if the process returns success, then all adaptive executions with finite steps of adaptations satisfy their corresponding transitional properties.

In the adaptive routing protocol, we found that the guarantee for state routed1 (LP_1) did not imply the condition $(\neg sent\, \mathcal{U}\, encrypted) \rightharpoonup LP_1$ in its assumption. This assumption condition required the obligation *encrypted* to be satisfied before the adaptation, while the guarantee did not ensure this obligation. Therefore, the model checking for the transitional property failed. As such, we generated a counterexample showing a path that violated the transitional properties by using a backtracking method. In this example, we returned the trace (ready2, received2, unsafe2, routed1). Clearly, the failure was caused by the adaptive transition a3 (from unsafe2 to routed1). We removed a3 from the adaptive program and re-performed steps (3') and (4'). The algorithm returned success.

7.4 Details of Model Checking Algorithms

In this section, we give the formal details of the model checking algorithms that we used in Section 7.3. The first algorithm checks whether a simple adaptive program satisfies its transitional property. The second algorithm extends the first algorithm in order to check the transitional properties of an n-plex adaptive program. The third algorithm checks the global invariants of an n-plex adaptive program. In all three algorithms, we use traditional LTL model checking to verify the base conditions.

The basic idea behind all three algorithms is to first calculate the assumptions and guarantees of each steady-state program using the marking algorithm, and store them in two interface structures I_1 and I_2, respectively. Then we compare I_1 and I_2 to determine whether the adaptive program satisfies the corresponding requirements.

7.4.1 Simple Adaptive Programs

We first introduce the modular model checking procedure for a simple adaptive program. Given a source program P_i, a target program P_j, an adaptation set $A_{i,j}$, a source local property ϕ_i, a target local property ϕ_j, and an adaptive constraint $\Omega_{i,j}$, the algorithm determines whether the adaptation from P_i to P_j through $A_{i,j}$ satisfies $\phi_i \overset{\Omega_{i,j}}{\rightarrow} \phi_j$, that is, the program changes from satisfying ϕ_i to satisfying ϕ_j.

ALGORITHM 1: Transitional properties for simple adaptive programs

input P_i, P_j: EFSM

input $A_{i,j}$: FSM

input $\phi_i, \phi_j, \Omega_{i,j}$: LTL

output ret: Boolean

local I_1, I_2: Interface

begin

1. Initialize two interfaces.

   $I_1 := \top$

   $I_2 := \top$

2. Verify programs P_i and P_j against properties ϕ_i and ϕ_j locally using traditional LTL model checking methods.
3. Construct marking $MARK''$ by running the marking algorithm on P_j with initial formula ϕ_j.
4. Calculate the state intersection tos_i of $A_{i,j}$ and P_j, where *tos* refers to *target of outgoing adaptation state.*

$$tos_i := S(A_{i,j}) \cap S(P_j). \tag{7.18}$$

5. Construct interface I_2 such that the conditions associated with states in tos_i are the same as their markings in $MARK''$, and the conditions associated with states

not in tos_i are *true*:

$$I_2 := \lambda(x : State).\ (\textbf{if } x \in tos_i \quad \textbf{then } MARK''(x) \quad \textbf{else true endif }). \tag{7.19}$$

6. Construct marking $MARK$ by running the marking algorithm on P_i with initial formula ϕ_i.
7. Calculate the state intersection sos_i of P_i and $A_{i,j}$, where *sos* refers to *source of outgoing adaptation state.*
8. Construct marking $MARK'$ by running the marking algorithm on $A_{i,j}$ with the initial formula mapping function Ψ as follows:
   - For each $s \in sos_i$, a formula $(x \stackrel{\Omega_{i,j}}{\rightarrow} \phi_j)$ is a conjunct of $\Psi(s)$ if and only if $x \in MARK(s)$.
9. Construct interface I_1 such that

$$I_1 := \lambda(x : State).\ (\textbf{if } x \in tos_i \quad \textbf{then } MARK'(x) \quad \textbf{else true endif }). \tag{7.20}$$

10. Compare I_2 and I_1 to see if I_2 implies I_1:

    $ret := I_2 \Rightarrow I_1$.

end

7.4.2 *N*-plex Adaptive Programs

This algorithm extends the algorithm for a simple adaptive program to a general n-plex adaptive program M. Given a set of steady-state programs P_i, a set of adap-

tation sets $A_{i,j}$, a set of local properties ϕ_i, and a set of adaptive constraints $\Omega_{i,j}$, the algorithm determines:

1. For all $i \neq j$, whether the adaptation from P_i to P_j through $A_{i,j}$ satisfies $\phi_i \overset{\Omega_{i,j}}{\rightarrow} \phi_j$. That is, the program changes from satisfying ϕ_i to satisfying ϕ_j.

2. Whether any execution from P_{j_1}, going through $P_{j_2}, P_{j_3}, \cdots, P_{j_k}$ $(j_i \neq j_{i+1})$ satisfies $\phi_{j_1} \overset{\Omega_{j_1,j_2}}{\rightarrow} \phi_{j_2} \cdots \phi_{j_k}$, i.e., sequentially satisfying $\phi_{j_1} \cdots \phi_{j_k}$.

This algorithm repeatedly applies **Algorithm 1** to each single adaptation from P_{j_1} to P_{j_k}. As some of the marking and comparison operations overlap, the algorithm is optimized by removing the redundancies.

ALGORITHM 2: Transitional property for n-plex adaptive programs

input P_i $(i = 1 \cdots n)$: EFSM

input $A_{i,j}$ $(i, j = 1 \cdots n)$: FSM

input $\phi_i, \Omega_{i,j}$ $(i, j = 1 \cdots n)$: LTL

output ret: Boolean

local I_1, I_2: Interface

begin

1. Initialize two interfaces.

   $I_1 := \top$

   $I_2 := \top$

2. For each program P_i

   (a) Verify programs P_i against properties ϕ_i locally with traditional LTL model checking methods.

   (b) Generate markings $MARK$ by running the marking algorithm on P_i with initial formula ϕ_i.

(c) Calculate tis_i, the target states of all incoming adaptive transitions, for all $j \neq i$, i.e., $tis_i = \bigcup_{j(j \neq i)}(P_i \cap A_{j,i})$, where *tis* refers to *target of incoming adaptation state*.

(d) Update interface I_2 with I_2' such that the conditions associated with states in tis_i are the conjunction of their values in I_2 and their markings in $MARK$, and the conditions associated with states not in tis_i are those in I_2:

$$I_2' := \lambda(x : State).\ (\textbf{if } x \in tis_i \quad \textbf{then } MARK(x) \wedge I_2(x) \quad \textbf{else } I_2(x) \textbf{ endif }). \tag{7.21}$$

(e) For each $j \neq i$

(i) Calculate the state intersection $sos_{i,j}$ of P_i and $A_{i,j}$.

(ii) Construct marking $MARK'$ by running the marking algorithm on $A_{i,j}$ with the initial formula mapping function Ψ as follows:

- For each $s \in sos_{i,j}$, a formula $(x \overset{\Omega,\alpha}{\rightarrow} \phi_j)$ is a conjunct of $\Psi(s)$ if and only if $x \in MARK(s)$.

(iii) Calculate the state intersection $tos_{i,j}$ of P_j and $A_{i,j}$.

(iv) Update interface I_1 with I_1' such that

$$I_1' := \lambda(x : State).\ (\textbf{if } x \in tos_{i,j} \quad \textbf{then } MARK'(x) \wedge I_1(x) \quad \textbf{else } I_1(x) \textbf{ endif }). \tag{7.22}$$

3. Compare I_2 and I_1 to see if I_2 implies I_1:

$ret := I_2 \Rightarrow I_1$.

end

7.4.3 Global Invariants

Given a set of steady-state programs P_i, a set of adaptation sets $A_{i,j}$, and a global invariant INV, the global invariant model checking algorithm determines whether all executions of an n-plex adaptive program satisfy the global invariant INV.

ALGORITHM 3: Global invariants

input P_i $(i = 1 \cdots n)$: EFSM

input $A_{i,j}$ $(i, j = 1 \cdots n)$: FSM

input INV: LTL

output ret: Boolean

local I_1, I_2: Interface

begin

1. Initialize two interfaces.

   $I_1 := \top$

   $I_2 := \top$

2. For each program P_i:

   (a) Verify programs P_i against global invariants INV with traditional LTL model checking methods.

   (b) Construct the program composition

   $$C_i = comp(P_i, union(A_{i,1}, A_{i,2}, \cdots, A_{i,n})). \tag{7.23}$$

   (c) Construct marking $MARK$ by running the marking algorithm on C_i with initial formula INV.

   (d) Calculate tis_i, the union of the target states of all incoming transitions, i.e., $tis_i = \bigcup_{j(j \neq i)}(P_i \cap A_{j,i})$

   (e) Update interface I_2 with I_2' such that the conditions associated with states in tis_i are the same as the conjunction of their values in I_2 and their markings in

$MARK$, and the conditions associated with states not in tis_i are the values in I_2:

$$I_2' = \lambda(x : State).\ (\textbf{if } x \in tis_i \quad \textbf{then } MARK(x) \wedge I_2(x) \quad \textbf{else } I_2(x) \textbf{ endif }). \tag{7.24}$$

(f) Calculate tos_i, the union of the state intersections of $A_{i,j}$ and P_j for all $j \neq i$, i.e., $tos_i = \bigcup_{j(j \neq i)}(A_{i,j} \cap P_j)$.

(g) Update interface I_1 with I_1' such that

$$I_1' = \lambda(x : State).\ (\textbf{if } x \in tos_i \quad \textbf{then } MARK(x) \wedge I_1(x) \quad \textbf{else } I_1(x) \textbf{ endif }). \tag{7.25}$$

3. Compare I_2 and I_1 to see if I_2 implies I_1:

$ret := I_2 \Rightarrow I_1$.

end

7.4.4 Claims

The following theorems capture the claims that the algorithms introduced in this section solve the problems they at intended to solve.

Note that the claims in this section apply to the eventuality of A-LTL and LTL properties as well.

Theorem 10:

*For a simple adaptive program from P_i to P_j, if **Algorithm 1** returns true, then:*

- *All non-adaptive executions within P_i (or P_j) satisfy the local property ϕ_i (or ϕ_j).*
- *All adaptive executions starting from P_i and adapting to P_j satisfy $\phi_i \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j$.*

Proof 10:

The claim that all non-adaptive executions within P_i (or P_j) satisfy the local property ϕ_i (or ϕ_j) is guaranteed by the traditional LTL model checking methods.

Let σ be an adaptive execution from P_i to P_j. Let $s_i \in S(P_i)$ be the first state in the adaptation and $s_j \in S(P_j)$ be the last state in the adaptation. Let ψ_i and ψ_j be the obligations for s_i and s_j, propagated though σ in markings M and M', respectively.

1. $\sigma^j \models \psi_j$.

   $I_2(s_j) \Rightarrow I_1(s_j) \Rightarrow \psi_j$, *therefore* $\sigma^j \models \psi_j$.

2. $\sigma^i \models \psi_i \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j$.

   There exists a finite path π in the property automaton PROPERTY$(\psi_i \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j)$ *that simulates $s_i, s_{i+1}, \cdots, s_j$, and $N(\pi_{j-1}) \Rightarrow \psi_j$. From Lemma 2, we have $\sigma^i \models \psi_i \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j$.*

3. $\sigma \models \phi_i \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j$.

   As illustrated in Figure 7.5, since $\sigma^i \models \psi_i \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j$, there exists s_k, such that $s_i, s_{i+1}, \cdots, s_k \models_{fin} \psi_i$ and $s_{k+1}, s_{k+2}, \cdots \models \phi_j$. Then we have $s_i, s_{i+1}, \cdots, s_k, s_k, s_k \cdots \models \psi_i$. From Lemma 2, we have

   $$s_0, s_{i+1}, \cdots, s_k, s_k, s_k, \cdots \models \phi_i,$$

   and therefore, $s_i, s_{i+1}, \cdots, s_k \models_{fin} \phi_i$. Thus, $\sigma \models \phi_i \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j$.

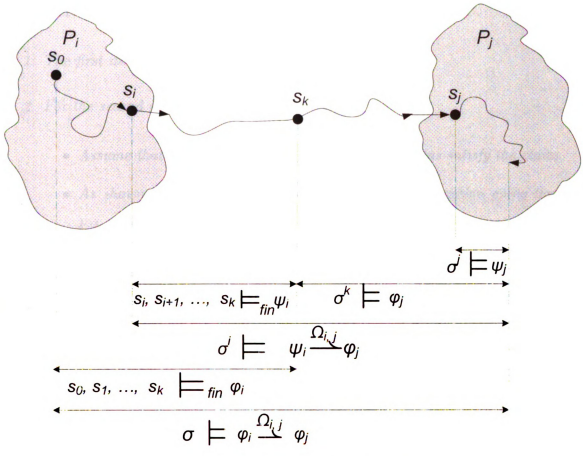


Figure 7.5: Illustration for Proof 10

Theorem 11: *For an n-plex adaptive program M, if* ***Algorithm 2*** *returns true, then:*

1. *All non-adaptive executions within a single steady-state program P_i satisfy the local property of P_i.*
2. *Any execution σ starting from P_{j_1} going through $P_{j_2}, \cdots P_{j_k}$ $(j_i \neq j_{i+1})$ satisfies*

$$\phi_{j_1} \overset{\Omega_{j_1,j_2}}{\sim} \phi_{j_2} \overset{\Omega_{j_2,j_3}}{\sim} \phi_{j_3} \cdots \overset{\Omega_{j_{k-1},j_k}}{\sim} \phi_{j_k}. \tag{7.26}$$

Proof 11:

1. *The first item is a direct conclusion from Theorem 10.*

2. *For the second item, we prove by induction.*

   - *Assume that all executions going through k programs satisfy the claim.*
   - *As shown in Figure 7.6, let $\sigma = s_0\ s_1, \cdots$ be an execution going through $k+1$ programs $P_{j_1}, P_{j_2}, \cdots, P_{j_{k+1}}$.*

     Let s_l and s_m be the states exiting P_{j_k} and entering $P_{j_{k+1}}$, respectively. Let σ' be a finite path in P_{j_k}, starting from an initial state $s_{j_k 0}$ of P_{j_k}, ending in s_l. Based on the assumption, we have

$$\sigma' \frown s_{l+1}, s_{l+2}, \cdots \models \phi_{j_k} \overset{\Omega_{j_k, j_{k+1}}}{\rightharpoonup} \phi_{j_{k+1}}. \tag{7.27}$$

     Therefore, there exists i $(l < i < m)$, such that $\sigma^i \models \phi_{j_{k+1}}$ and $\sigma' \frown s_{l+1}, s_{l+2}, \cdots, s_i \models_{fin} \phi_{j_k}$.

     Any path starting in P_{j_i} ending in P_{j_k} satisfies

$$\phi_{j_1} \overset{\Omega_{j_1, j_2}}{\rightharpoonup} \phi_{j_2} \overset{\Omega_{j_2, j_3}}{\rightharpoonup} \phi_{j_3} \cdots \overset{\Omega_{j_{k-1}, j_k}}{\rightharpoonup} \phi_{j_k}. \tag{7.28}$$

     Thus,

$$s_0, s_1, \cdots s_i \models_{fin} \phi_{j_1} \overset{\Omega_{j_1, j_2}}{\rightharpoonup} \phi_{j_2} \overset{\Omega_{j_2, j_3}}{\rightharpoonup} \phi_{j_3} \cdots \overset{\Omega_{j_{k-1}, j_k}}{\rightharpoonup} \phi_{j_k}. \tag{7.29}$$

     Therefore,

$$\sigma \models \phi_{j_1} \overset{\Omega_{j_1, j_2}}{\rightharpoonup} \phi_{j_2} \overset{\Omega_{j_2, j_3}}{\rightharpoonup} \phi_{j_3} \cdots \overset{\Omega_{j_k, j_{k+1}}}{\rightharpoonup} \phi_{j_{k+1}}. \tag{7.30}$$

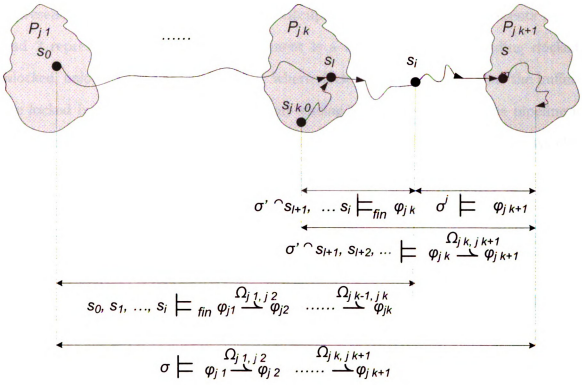


Figure 7.6: Illustration for Proof 11

Theorem 12: *For an n-plex adaptive program M, if* ***Algorithm 3*** *returns true, then all execution paths of M satisfy the global invariant INV.*

Proof 12:

This proof is similar to that for the transitional properties.

7.5 Case Study: Adaptive Java Pipeline Program

In this section, we demonstrate the performance improvement of our approach over existing approaches by applying it to a more complex case study: the adaptive Java pipeline program [151] introduced in Section 5.4.

Figure 7.7 shows the simplified adaptive Java pipeline state machine (with 2 data buffers). The label of each state is a pair: The first element (ranging from 0 to 2)

represents the number of buffers already occupied by data (where 0 represents *empty* and 2 represents *full*); the second element is a *status indicator* including **rlocked**, **wlocked**, **unlocked**, **reading**, and **writing**, where **rlocked/wlocked** means that the buffers are locked by a reader/writer thread, and **reading/writing** means that the pipeline is being read/written by a reader/writer thread. Transitions are labeled with **wlock?**, **wlock!**, **rlock?**, **rlock!**, **w?**, **w!**, **r?**, and **r!**, where **rlock?/wlock?** represents acquiring a read/write lock, and **rlock!/wlock!** represents releasing a read/write lock. **r?/w?** represents starting a read/write operation, and **r!/w!** represents completing a read/write operation.

The upper rectangle in Figure 7.7 illustrates P_1, the synchronized Java pipeline. In the initial state **(0,unlocked)**, the buffer is empty and unlocked. The buffers can be locked by the reader/writer, and the program goes to **(0,rlocked)/(0,wlocked)**. When the buffer is not *empty/full*, and a rlock/wlock is set, a read/write operation can be performed, and the operation must not interleave with any other operations until it completes. The lower rectangle in Figure 7.7 shows the asynchronous Java pipeline state machine. In this version, no lock is required, and the reader/writer can start reading/writing while the other is still in action. In the state $(1, \langle writing, reading \rangle)$, both the write and read operations are active. We define two adaptive transitions: α and β connecting the two steady-state programs. Although more adaptive transitions are possible, for brevity, we only discuss this pair of adaptive transitions in this chapter. The global invariants are as follows:

- **Buffer integrity**: The program must not read/write while the buffer is *empty/full*.

$$inv_1 = \Box(empty \Rightarrow \neg r?).$$

$$inv_2 = \Box(full \Rightarrow \neg w?).$$

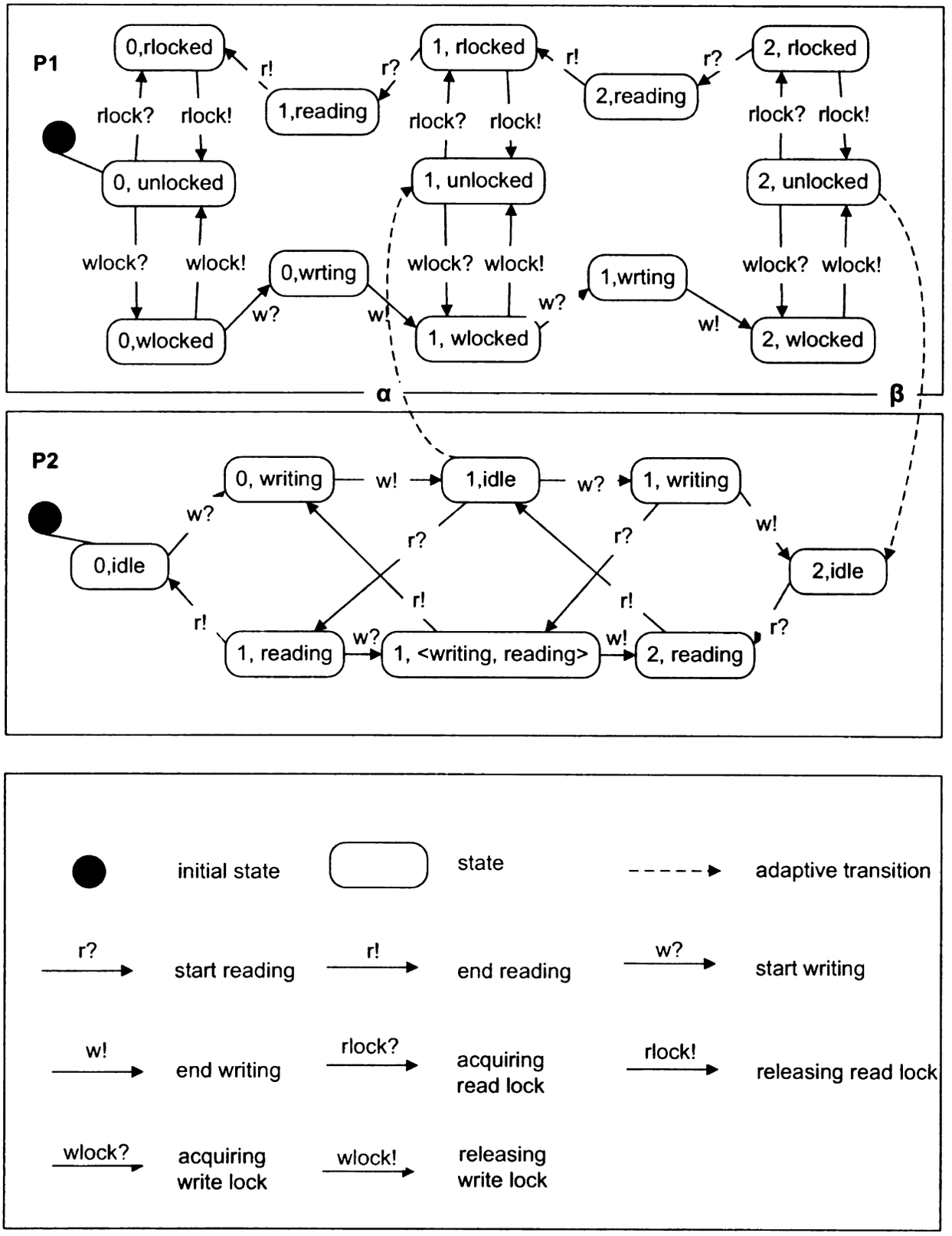


Figure 7.7: Case study: adaptive Java pipeline

- **Lock integrity**: After locking (rlock?/wlock?) the buffer, the reader/writer should eventually unlock (rlock!/wlock!) it before it can be locked by the other type of lock.

$$inv_3 = \Box(rlock? \Rightarrow \neg wlock? \,\mathcal{U}\, rlock!).$$
$$inv_4 = \Box(wlock? \Rightarrow \neg rlock? \,\mathcal{U}\, wlock!).$$

- **Read/write integrity**: After the program starts reading/writing, it should complete reading/writing before the next reading/writing starts.

$$inv_5 = \Box(r? \Rightarrow \bigcirc(\neg r? \,\mathcal{U}\, r!)).$$
$$inv_6 = \Box(w? \Rightarrow \bigcirc(\neg w? \,\mathcal{U}\, w!)).$$

We have successfully verified all of the above properties with the AMOEBA model checker. In order to study the scalability of our approach, we synthesized a series of n-plex adaptive programs by duplicating the above steady-state programs $n/2$ times (where n ranges from 2 to 200), and connecting these duplicates with adaptive transitions. We measured the execution time and memory consumption and compared the results to the monolithic and the pairwise implementations introduced in Section 7.1. The experimental results were evaluated on a dual processor 800MHZ Pentium III SMP with 32KB L1 and 256KB L2 caches, and 256MB main memory.

running Redhat Linux 7.2 (kernel version: Linux 2.4 smp). Figure 7.8(a) shows the experimental results of the average execution time of model checking for the above 6 global invariants, where the x-axis represents the number of steady-state programs in the adaptive programs and the y-axis represents the time used for executions. The assessment of the results indicates that the time consumed by our approach is linear to the number of steady-state programs. It is about n times faster than the pairwise approach, and slightly slower than the monolithic approach. Figure 7.8(b) shows the memory usage where the x-axis represents the number of steady-state programs in the adaptive programs and the y-axis represents the memory occupied by the programs in the numbers of states stored by the algorithms during model checking. The assessment of the results indicates that our approach uses almost constant amount of memory, which is approximately $1/n$ of the amount of memory required by the monolithic approach and 1/2 of the memory required by the pairwise approach. The above results conform to the theoretical complexity analysis discussed in Section 7.6.2.

7.6 Optimizations, Scalability, and Limitations

Next, we discuss performance issues including optimizations, scalability, and limitations.

7.6.1 Optimizations

In our algorithms, we need to compare the guarantee interface with the assumption interface to determine whether the guarantees imply the assumptions. The guarantee and assumption of each state are both sets of A-LTL formulae. The implication between guarantees and assumptions is the implication between the conjunctions of the formulae, i.e.,

$\bigwedge(G_i) \Rightarrow \bigwedge(A_i) \equiv true$, where G_is are conditions in the guarantee and A_is are

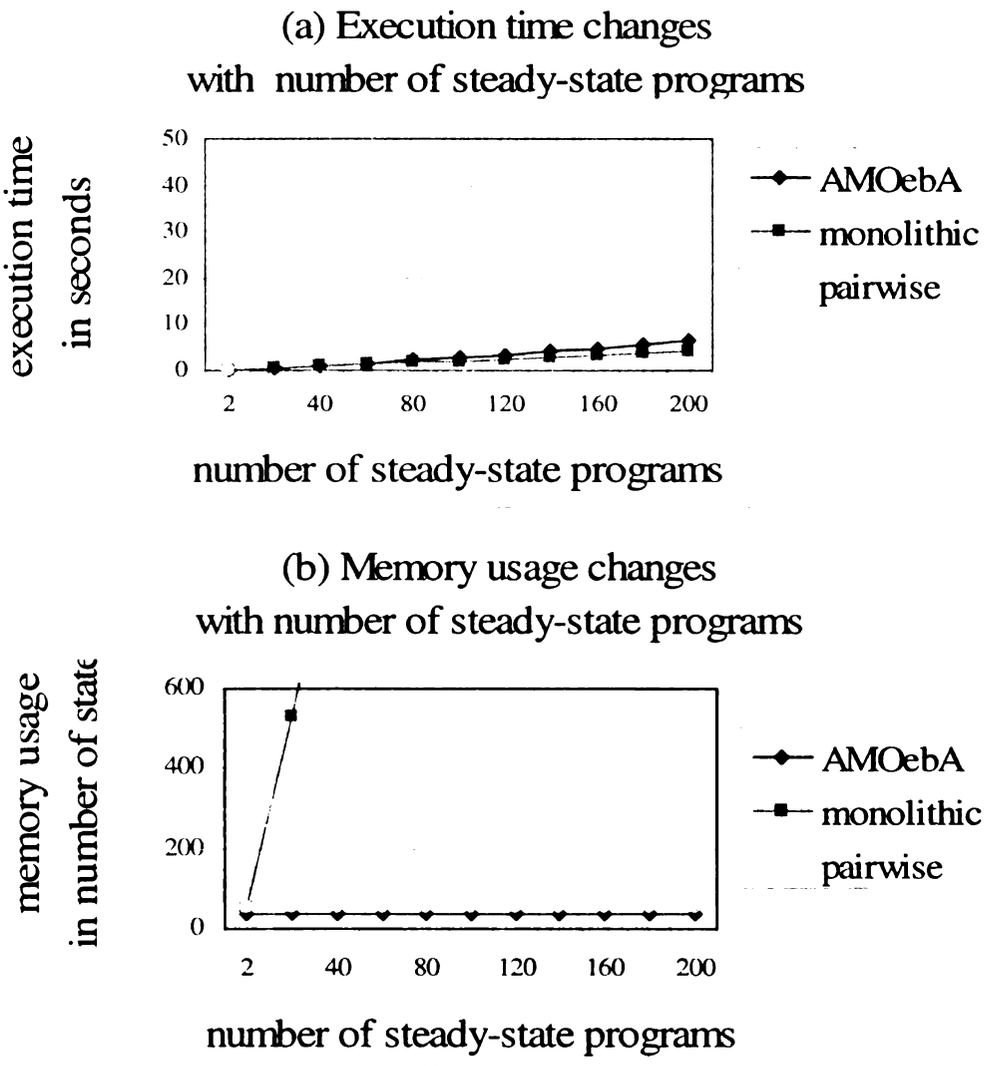


Figure 7.8: Performance comparison between our approach and alternative approaches

conditions in the assumption.

$$\bigwedge_j (G_j) \Rightarrow \bigwedge_i (A_i) \equiv true \quad \Leftrightarrow \quad \bigwedge_i \bigvee_j G_j \Rightarrow A_i \equiv true. \tag{7.31}$$

Its sufficient condition is

$$\bigwedge_i (\bigvee_j G_j \Rightarrow A_i \equiv true). \tag{7.32}$$

Therefore we turn the comparison between conjunctions of formulae into the

comparison between two formulae, where each formula is the next formula of a node in the property automaton. To improve performance, we check the sufficient condition instead of the original formula. This is another reason why our approach is sound but incomplete.

The algorithm introduced in Section 7.3 stores the guarantee and assumption markings of all the states in the adaptive program, which may occupy a large amount of memory. However, in later steps, only markings of a small portion of the states are used, i.e., the states with incoming and outgoing adaptive transitions. We call these states *interface states*, and the assumption/guarantee markings of these states *assumption/guarantee interfaces*, respectively. In the AMOEBA implementation, the required memory space is significantly reduced by storing the markings for only the interface states instead of for all the states during the marking computations.

Our approach can also be further optimized to verify incrementally developed software by utilizing existing verification results. Assume that an n-plex adaptive program has been successfully verified with our approach. When a steady-state program P_i of the adaptive program is changed after a successful model checking has been performed, we only need to repeat **Algorithm 2** and **Algorithm 3** on P_i (and/or related adaptations) to determine whether the specifications are still satisfied. Assume that if after reapplying marking algorithm on P_i, we compute the interfaces to be I_1' and I_2', then

- if $I_2' \Rightarrow I_2$ and $I_1 \Rightarrow I_1'$, then we have $I_2' \Rightarrow I_1'$ (since implicitly, $I_2 \Rightarrow I_1$), then no more model checking is required;
- if $I_1 \not\Rightarrow I_1'$, then the model checking for outgoing adaptations from P_i needs to be repeated;
- if $I_2' \not\Rightarrow I_2$, then the model checking for incoming adaptations to P_i needs to be repeated.

The model checking results for all other parts still apply and therefore are reused.

When a new steady-state program P_{n+1} is incrementally introduced to an n-plex adaptive program after the adaptive program has been successfully verified with our algorithms, we will need to model check all the adaptations going into and out of P_{n+1}. Incremental model checking significantly reduces the model checking overhead in incrementally developed software.

7.6.2 Complexities and Scalability

Our proposed modular model checking approach increases scalability of model checking by reducing the time/space complexities of the model checking algorithms. We explain this point by comparing our approach to the alternative approaches described in Section 7.1, namely, the pairwise approach [2] and the monolithic approach [27, 93]. Assume an n-plex adaptive program M contains steady-state programs $P_1, P_2, \cdots, P_n$. We denote the size of the steady-state program P_i as $\mid P_i \mid$, and we assume all steady-state programs are of similar size $\mid P \mid$: $\mid P \mid \approx \mid P_i \mid$, for all i.

We assume that the size of adaptive states and transitions is significantly smaller than that of the steady-state programs (which we consider a key characteristic of adaptive software). Then we have $\mid M \mid \approx n * \mid P \mid$, where $\mid M \mid$ is the size of M.

The complexity of the global invariant model checking is the sum of the complexity of each step introduced in Section 7.3. The most time/space consuming step is computing the assumption and guarantee interfaces. We can prove that its time complexity is $O(2^{|INV|} \mid M \mid)$ and its space complexity is $O(2^{|INV|} \mid P \mid)$. The time and space complexities of the monolithic approach are both $O(2^{|INV|} \mid M \mid)$. That is, while the monolithic approach requires the same order of time, it requires n times more memory space than our approach. The time and space complexities of the pairwise approach are $O(n(2^{|INV|} \mid M \mid))$ and $O(2^{|INV|} \mid P \mid)$, respectively. That is, while

it requires the same order of space, it requires n times more execution time than our approach.

The time and space complexities of our transitional property model checking algorithm are $O(2^{|LP|} \mid M \mid)$ and $O(2^{|LP|} \mid P \mid)$, respectively. To the best of our knowledge, no alternative approach solves the same problem. The approach that handles the closest problem is the ITL model checking algorithm proposed by Thompson and Bowman [17], which can be applied to verify that executions of only a specific adaptation sequence satisfy their corresponding transitional property, instead of all possible adaptation sequences as with our approach. The memory consumption of their approach is linear to the size of the program automaton, which implies that our approach is at least n times more efficient in terms of space. In addition, their approach is non-elementary to the length of the equivalent A-LTL formulae,[4] while our approach is exponential, which means that our approach is at least exponentially more efficient.

Based on the above analysis, we conclude that our approach reduces the verification space/time complexity by a factor of n, and it is more scalable for verifying incremental changes to the adaptive program at development time.

7.6.3 Limitations

The following assumption must be considered when applying our approach: We regard the essential characteristic of an adaptive program to be that the n steady-state programs are loosely coupled, i.e., the size of adaptive states and transitions are significantly smaller than the size of the steady-state programs. This assumption is reasonable since it is generally desirable software engineering practice to minimize the coupling among program modules [104]. Our study of the adaptive Java pipeline

[4]The level of exponents is linear to the length of the formula, i.e., $2^{2^{2^{\cdots^{2}}\}|L|}}$.

example supports this assumption: The size of adaptive states and transitions consistently accounts for 1% of the size of the overall adaptive program. Similar phenomena are also observed in other adaptive programs that we have developed for the RAPIDware project. More thorough studies are required to validate this assumption in other adaptation scenarios. This assumption ensures that both the number of markings that we need to store and the comparisons between interface markings that we need to perform are minimal.

Although our approach is sound, it is incomplete, i.e., the algorithm may return a counterexample that actually satisfies the properties under verification. We consider the false positives to be an indication of lack of clarity in the specification. The false positives can be discharged by more clearly specifying the adaptive program.

7.7 Related Work

In this section, we analyze the differences and relationships between our approach and other modular verification approaches.

Krishnamurthi *et al* [74] introduced a modular verification technique for *aspect-oriented* programs. They model a program as a finite state machine (FSM), where an *aspect* is a mechanism used to change the structure of the program FSM. An aspect includes a *point-cut designator*, which defines a set of states in the program, an *advice*, which is an FSM itself, and an *advice type*, which determines how the program FSM and the advice FSM should be composed. By model checking the program and the aspect individually, they verify the Computation Tree Logic (CTL) properties of the composition of the program and the aspect. In other work [44, 86], they introduced modular model checking techniques for cross-cutting features, which basically follow the same idea. Our approach is largely inspired by their approach. The two approaches share the following similarities: (1) Both have goals to decompose the

model checking of a large system into the model checking of smaller modules. (2) We both use the assume/guarantee reasoning first formulated by Misra and Chandy [101] in 1981, while neither of us invented the concept. (3) We both define interface states and use conditions to mark these states. Then we reason about the conditions among these states to draw conclusions about the entire system. (4) There are behavioral changes involved in both approaches, although in different ways. (5) Neither of our approaches is complete.

However, despite the similarities, the two approaches also have significant differences, and their approach is not applicable to our verification problems. (1) The most prominent difference is the fundamental difference between the underlying structures for the temporal logics that we handle. CTL is evaluated on states. The classic model checking method for CTL is accomplished by using state marking [38]. Their approach reused the classic CTL state marking algorithm on interface states, which can be considered a natural extension of the basic CTL model checking idea. However, LTL/A-LTL is evaluated on execution paths. Model checking of LTL **cannot** be solved by marking its states. Rather, tableau-based and automaton-based methods are conventionally used for path-based logic model checking. The challenge overcome by our approach is to design a novel marking algorithm to be applied to states for a path-based logic, which has not been previously published in the literature [145]. Our algorithm also deals with eventuality, which is also a challenge for LTL/A-LTL, but not for CTL model checking. (2) Our approaches also differ in the way the system behavior may change. With their approach, one may consider the behavioral change from the base to the feature as an adaptation. They require the execution to change back to the base, which behaves analogously to a stack. In contrast, our approach allows arbitrary adaptation sequences, such as an adaptation through a sequence of programs A to B to C, etc. (3) Our approaches differ in the definition of modules. In their approach, a module refers to a separate piece of code (physically), such as the

base program or a feature. However, in our approach, each module refers to different behaviors. It may be exhibited by the same piece of code under different modes, or different pieces of code. (4) Last but not least, our approach supports the verification of A-LTL, which has not been previously addressed.

Henzinger *et al* [56] proposed the Berkeley Lazy Abstraction Software verification Tool (BLAST) to support extreme verification (XV). XV is modeled to be a sequence of programs and specifications (P_i, Φ_i), where Φ_i is the specification for the i^{th} version of the program P_i, and Φ_i are non-decreasing, i.e., $\Phi_i \subseteq \Phi_{i+1}$. In order to reduce the cost of each incremental verification when verifying the i^{th} program, they generate an *abstract reachability tree* T_i. When model-checking P_{i+1}, they compare P_{i+1} to T_i to determine the part of P_{i+1}, if any, should be re-verified. Our approach differs from their approach in that they verify propositional, instead of temporal logic properties. Also, their approach is for general programs, while our incremental verification is optimized specifically for adaptive programs. We consider their approach to be complementary to ours because, in practice, each steady-state program is developed incrementally from some common base program. Their approach can verify the local properties and global invariants of each steady-state program locally for the base condition verification.

Alur *et al* [4] introduced an approach to model-checking a *hierarchical state machine*, where higher-level state machines contain lower-level state machines. As the same state machine may occur at different locations in the hierarchy, its model checking may be repeated if we *flatten* the hierarchical state machine before applying traditional model checking. Their objective is to reduce the verification redundancy when a lower-level state machine is shared by a number of higher level state machines. Their approach can be applied to optimize our solution in that the steady-state programs may share parts of their behavior (sub-state machines). We can use their approach to reduce the redundancy when verifying the shared behavior.

Many others, including Kupferman and Vardi [80], Flanagan and Qadeer [46], have also proposed modular model checking approaches for different program models. They focus on verifying concurrent programs, where modules are defined to be concurrent threads (processes). Their approaches essentially address a different perspective of model checking problems (e.g., data sharing among threads), and we consider their approaches to be orthogonal and complementary to our approach.

7.8 Discussion

In this chapter, we introduced a sound modular approach to model-checking adaptive programs against their global invariants and transitional properties expressed in LTL/A-LTL. For an n-plex adaptive program, given that the size of the adaptive states and transitions is usually significantly smaller than the size of the overall adaptive program, our approach is at least n times more efficient than alternative approaches. Also, our approach is applicable to the transitional properties which, previously, had not been analyzable with other approaches. Our approach is highly scalable in that the time complexity is linear to the number of steady-state programs, and the space complexity is linear to the size of each steady-state program. More performance improvement can be achieved by using our approach to incrementally verify adaptive programs. For validation purposes, we have implemented our approach in a prototype model checker AMOEBA using C++, and used the tool to verify a number of adaptive programs.

The AMOEBA model checker still suffers from the state explosion problem of static model checking, and may not be sufficient to be directly applied to an adaptive system if (1) the size of a single steady-state program exceeds the capacity of the model checker, or (2) the steady-state programs are not loosely coupled, and thus the size of the interface states exceeds the capacity of the model checker. In order to

address these cases, we have investigated run-time verification of adaptive programs against their global invariants and transitional properties, which will be introduced in Chapter 8.

As described in Section 7.7, our approach may be combined with existing techniques [2, 4, 44, 46, 56, 58, 72, 74, 80, 86] to further improve the performance of model checking adaptive software. In general, because our technique is complementary to many existing model checking techniques, investigating the combinations of our approach with these other techniques may yield interesting model checking capabilities for adaptive and non-adaptive systems.

Chapter 8

Run-Time Model Checking for Adaptive Software

In this chapter, we propose a run-time monitoring and verification technique to guarantee that executing adaptive software satisfies its adaptation requirements specified in A-LTL/LTL. In Chapter 7, we introduced the AMOEBA static model checker [145] to modularly verify A-LTL properties in adaptive software. However, despite the effort, AMOEBA still suffers from the state explosion problem and remains insufficient to be directly applied to certain types of adaptive software, including those with exceedingly large and tightly coupled steady-state programs. Run-time verification is an attractive complement to static verification as a means to verify formal properties in adaptive software.

Run-time model checking monitors executions of a software system and verifies them against a set of formal specifications, all at run time. Since only one execution path is examined at a time, the state explosion problem is largely avoided. Existing run-time model checking projects include Java PathExplorer (JPaX) [55], the run-time monitoring and checking (MaC) architecture [83], temporal rover tools [35], EAGLE [9], etc. However, these projects do not verify A-LTL properties for adaptive

software.

We introduce AMOEBA-*RT*, a run-time A-LTL model checker for adaptive software. In AMOEBA-RT, the run-time state information of an adaptive program is collected by instrumenting the program. Since the instrumentation is a crosscutting concern [21], we use an aspect-oriented approach [132] to capture this concern in an aspect file. The aspect-oriented approach is non-invasive, meaning that the source code for the adaptive software is not directly altered. At run time, the instrumentation code sends the collected state information to a run-time model checking server, running as a separate process. The run-time model checking server uses an automaton-based approach to determine whether the state information received from the adaptive program satisfies adaptation properties specified in A-LTL.

AMOEBA-RT is a prototype system for run-time model checking of adaptive software, where an aspect-oriented technique [132] is used to instrument the software for analysis. We illustrate AMOEBA-RT with the adaptive Java pipeline program introduced in Chapter 5.

The remainder of this chapter is organized as follows. Section 8.1 briefly introduces the AMOEBA-RT model checker. In Section 8.2, we illustrate the run-time verification using the adaptive Java pipeline example. Related work is overviewed in Section 8.3, and Section 8.4 summarizes our approach and discusses limitations and extensions.

8.1 Run-Time Verification

AMOEBA-RT is a technique for the verification of A-LTL properties in adaptive software. AMOEBA-RT is an extension of the AMOEBA model checker introduced in Chapter 7, extended with the support of run-time monitoring and run-time verification. In general, run-time verification monitors the run-time behavior of a software

system and checks its conformance to a requirements specification defined as a temporal logic property [9]. Run-time model checking can be considered a combination of testing and model checking as an effort to achieve the benefit of both approaches, and meanwhile, to avoid the pitfalls of *ad hoc* testing and the complexity of model checking [55]. During the execution of the software, an execution trace is generated, which is fed to a model checker to verify its conformance to the formal specification. A common implementation of the model checker is to use a property automaton running in parallel with the adaptive software. Given a formal specification, the property automaton is constructed in such a way that it accepts exactly the set of executions that satisfy the specification. Therefore, the model checker accepts the execution trace if and only if the trace satisfies the specification.

The two key extensions in AMOEBA-RT are the run-time monitoring of the executing adaptive software achieved by an aspect-oriented instrumentation technique, and the run-time verification of the A-LTL/LTL adaptation specifications achieved by a run-time model checking server. As shown in Figure 8.1, AMOEBA-RT includes a program instrumentation module and a run-time model checking server module.

8.1.1 Aspect-Oriented Instrumentation

As shown in Figure 8.1, the instrumentation module instruments adaptive software with instrumentation code that collects run-time state information in the software and transmits the information to the run-time model checking server. The instrumentation is achieved by using an aspect-oriented technique [132], where the source code for the program to be verified is not altered directly. The AspectJ compiler compiles the Java source files and an aspect file specifying the instrumentation, and generates instrumented Java bytecode files. For example, the following aspect code inserts an evaluation of the variable `bar` after every time it is changed, and sends a condition ``high'' or ``low'' to the `checker` based on its comparison with

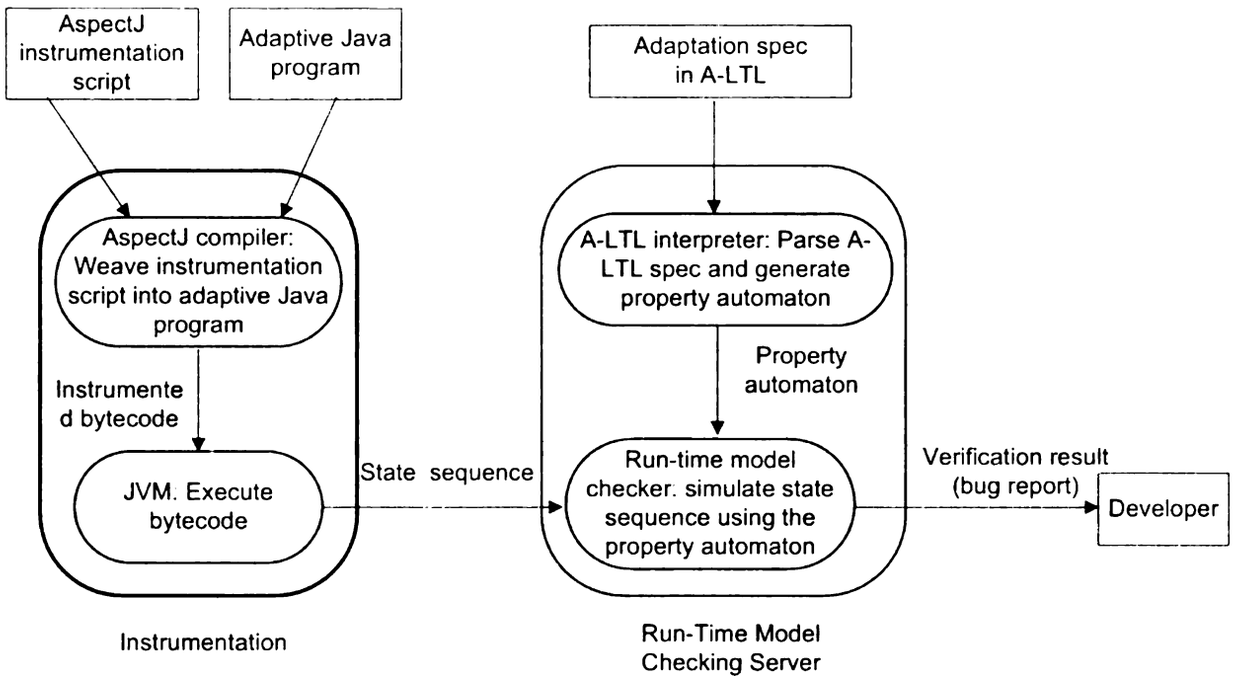


Figure 8.1: The dataflow diagram for AMOEBA-RT verification

a `threshold`.

```
public pointcut BAR(): set (int bar);
after(): BAR(){
    if( bar > threshold)
        checker.send ("high");
    else
        checker.send("low");
}
```

The Java bytecode files are then executed on a general JVM. During run time, the instrumentation code collects run-time state information and sends the information to the run-time model checking server in a sequence through an inter-process/host communication channel, such as remote procedure calls, etc. When the adaptive program terminates, an "*end of execution*" message (*EOE*) is attached to the end of the sequence and sent to the run-time model checking server by calling

```
checker.send("EOE");
```

8.1.2 Run-Time Verification

As shown in Figure 8.1, the run-time model checking server checks the conformance of the sequence of state information received from the instrumentation module with the adaptation requirements specified in A-LTL/LTL. An A-LTL interpreter processes the adaptation requirements specifications in A-LTL/LTL, and outputs a property automaton, i.e., a finite state automaton that accepts the exact set of execution paths satisfying the specification. The property automaton construction algorithm is the same as that used in the AMOEBA model checker (Section 7.2.2). However, instead of constructing a product automaton as described in Section 7.2.3, we use the property automaton to simulate the sequence of run-time state information received from the instrumentation module, in parallel with the adaptive software. The simulation process is described as follows: We use a variable curstate to denote the current state of the property automaton, initialized with the initial state of the property automaton. Upon the receipt of a condition cond from the instrumentation module, we invoke a `moveNext(cond)` method of the property automaton, which does the following:

- If $cond = EOE$ and curstate is an accepting state, then return success.
- If $cond = EOE$ and curstate is not an accepting state, then return failure.
- If $cond \neq EOE$ and curstate has a next state that agrees with the condition cond, then move curstate to the next state and return success.
- If $cond \neq EOE$ and curstate does not have a next state that is agrees with the condition cond, then return failure.

If the property automaton returns failure during or at the end of an execution, then the execution violates the A-LTL property. When a violation of the property automaton is encountered, the run-time model checking server module returns *false*, and the state sequence (i.e., a counterexample) is recorded in a bug report. Otherwise, the model checking server returns *true*.

8.2 Case Study: Adaptive Java Pipeline Program

We use the adaptive Java pipeline program introduced in Section 5.4 as an example to illustrate the run-time model checking technique. The architecture of the adaptive Java pipeline is shown in Figure 8.2. A writer thread contains an adaptive piped output component. The writer writes data to the adaptive piped output through a write interface. The adaptive piped output includes a sync piped output component and an async piped output component. It delegates write requests received from the writer to either one of the piped output components based on its current configuration. The writer configurations can be switched between synchronized and asynchronous modes by adaptation requests received from its adapt interface. Similarly, a reader thread contains an adaptive piped input component. The reader reads data from the adaptive piped input through a read interface. The adaptive piped input includes a sync piped input component and an async piped input component. It delegates read requests received from the reader to either one of the piped input components based on its current configuration. The reader configuration can be changed by adaptation requests received from its adapt interface. The data transmission from the output to the input is achieved by accessing shared buffers between the output and the input. A sync buffer and an async buffer are used for the synchronized and asynchronous pipeline components, respectively. Adaptations of the adaptive piped input/output components are coordinated by an adaptation driver component.

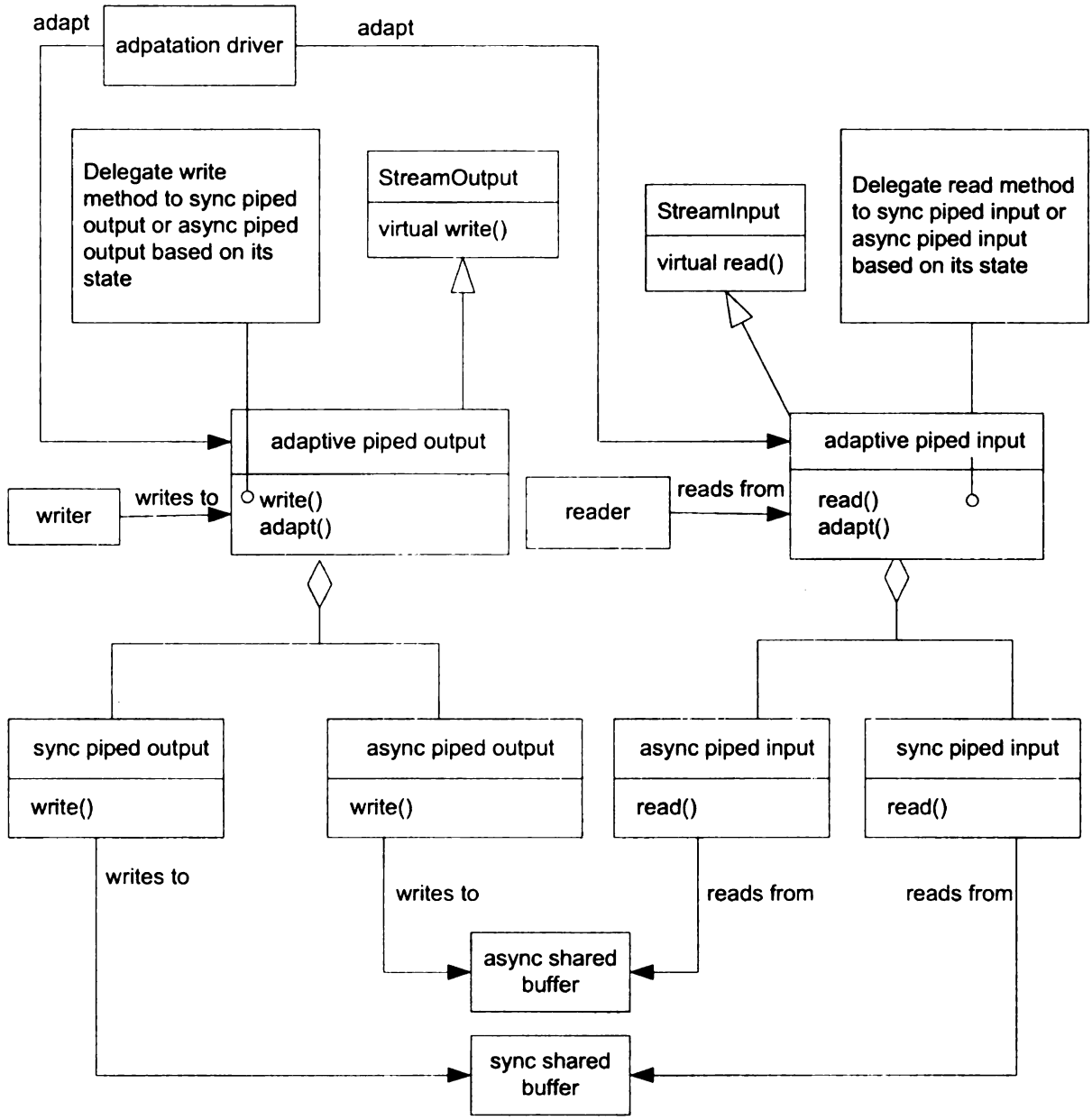


Figure 8.2: The adaptive Java pipeline

We illustrate the interactions among these components during adaptation with a sequence diagram in Figure 8.3. The reader and the writer threads are omitted for brevity. In the figure, bold message lines are for adaptation control messages, and thin message lines are for data read/write messages. For convenience, we separate our discussion of the sequence diagram into seven different phases, marked on the right-hand side of the figure.

- **Phase 1** illustrates the system behavior before adaptation, where the adaptive piped input and output components are both running in the synchronized mode. The adaptive input/output components delegate the read/write requests to the synchronized input/output components, which consequently, access the sync buffer.

- **Phase 2** shows the system behavior when the adaptation driver receives an adaptation request, adapt req. The adaptation driver sends an adapt output request to the adaptive piped output, which then terminates the sync piped output and starts the async piped output.

- In **phase 3**, after the adaptive piped output has adapted to the asynchronous mode, write requests to the adaptive piped output are delegated to the async piped output, and consequently, the data is written to the async buffer.

- In **phase 4**, after the adaptive piped output completes its adaptation, the adaptation driver sends an adapt input request to the adaptive piped input, which then attempts to terminate the sync piped input. However, the request cannot be executed immediately, since the sync piped input must clear the contents in the sync buffer before it terminates, otherwise, data in the buffer will be lost. Therefore, future read requests are still delegated to the sync piped input.

- In **phase 5**, upon receiving a read request read req, if the sync buffer is empty,

the sync piped input returns a stop ack message to the adaptive piped input, and terminates.

- In **phase 6**, the adaptive piped input starts the async piped input and returns adapt done to the adaptation driver. The adaptation driver returns adapt done to its run-time environment, thus completing the adaptation process.

- In **phase 7**, the read req is re-sent to the async piped input, which accesses data in the async buffer.

8.2.1 Adaptation Requirements

We specify the adaptation requirements for the adaptive Java pipeline program in A-LTL as follows. Before adaptation, the system (i.e., the source program) is required to input data from the synchronized pipeline in response to the outputs. That is, for each output data x in the synchronized mode, the system must eventually input data x. In LTL:

$$\Box(SyncOutput(x) \Rightarrow \Diamond SyncInput(x)). \tag{8.1}$$

The program behavior after adaptation can be specified in a similar manner. The system (i.e., the target program) is required to input data from the asynchronous pipeline in response to the outputs. In LTL:

$$\Box(AsyncOutput(x) \Rightarrow \Diamond AsyncInput(x)). \tag{8.2}$$

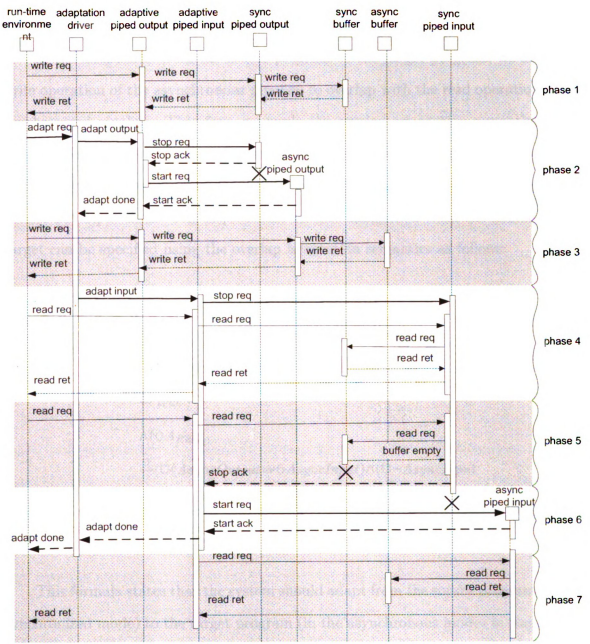


Figure 8.3: The sequence diagram for the adaptive Java pipeline

For both the synchronized and asynchronous pipelines, when an **output** event occurs, an input obligation is generated. The obligation will be fulfilled and discharged by a subsequent **input** event. Formulae (8.1 and 8.2) state that an execution must fulfill all input obligations before it terminates.

In the adaptation from the source program to the target program, we allow the **write** operation of the asynchronous pipeline to overlap with the **read** operation of the synchronized pipeline. Therefore, we apply the overlap adaptation semantics introduced in Section 3.3 to the specification of the adaptation. During the overlapped period, the synchronized pipeline should not output data, and the asynchronous pipeline should not input data. The requirement for the adaptation from the source to the target can be specified using the overlap adaptation semantics as follows:

$$
\begin{aligned}
&((\Box(SyncOutput \Rightarrow \Diamond SyncInput) \wedge (\Diamond A_{REQ} \\
&\overset{\Omega}{\rightarrow} \Box \neg SyncOutput)) \\
&\overset{\Omega}{\rightarrow} true) \\
&\wedge (\Diamond A_{REQ} \\
&\overset{\Omega}{\rightarrow} (\Box(AsyncOutput \Rightarrow \Diamond AsyncInput) \wedge (\Box \neg AsyncInput \\
&\overset{\Omega}{\rightarrow} true)))).
\end{aligned}
\tag{8.3}
$$

This formula states that the system should adapt from the source program (in the synchronized mode) to the target program (in the asynchronous mode) in response to the adaptation request A_{REQ}. The source and target programs overlap. During the overlapped period, the source must not output data, and the target must not input data. The output obligation generated in the synchronized mode must be fulfilled before the adaptation completes. In our adaptive Java pipeline design, this obligation

fulfillment requirement is achieved by the buffer empty check in the sync piped input component.

8.2.2 Instrumentation and Model Checking

In order to monitor the run-time execution conditions of the adaptive Java pipeline program, we use the aspect-oriented programming tool, AspectJ [132], to insert instrumentation code into the program. Currently, the AspectJ script for instrumentation is generated manually.

In this example, the "instrumentation concern" is encapsulated in an `Instrumentation` aspect, saved in a file named `Instrumentation.aj`. As shown in Figure 8.4, we define a pointcut `Main` to identify the `main()` method of the adaptive Java pipeline program (lines 1–2). In a *before advice* for the `Main` pointcut, i.e., at the very beginning of the entire program, we insert code to initialize the property automaton in the run-time model checking server by sending an A-LTL formula to the server (lines 6–16). The `AmoebaChecker` class implements a stub that is responsible for the communication with the model checking server. Its constructor method takes three parameters: The first two parameters specify the IP address and the port number for the model checking server, respectively. The third parameter specifies the A-LTL property to be verified. In an after advice of the `Main` pointcut, i.e., at the very end of each execution, we insert code to send an "end-of-execution" message to the run-time model checking server to terminate the model checking (lines 18–20).

Second, we use pointcuts to identify the locations of the adaptive Java pipeline program at which the sync and the async buffers are accessed. Figure 8.5 (a) illustrates the pointcut definition for the `SyncOutput` message. Line 2 defines that the point is within the `receive()` method of the synchronized input class. Line 3 defines that the point is at the location where the `buffer` is accessed. When the buffers are accessed for read/write, an input/output message will be generated and sent to

```
01  public pointcut Main():
02        execution(* mypackage.main(..));
03
04
05
06  before():Main(){
07    AmoebaChecker.checker =
08      new AmoebaChecker ("192.168.1.101",2211,
09          "((([](SyncOutput-><>SyncInput) /\\(<>AREQ
10           _>[]!SyncOutput))
11           _>true)
12           /\\ (<>AREQ
13           _> ([](AsyncOutput-><>AsyncInput)/\\([]!AsyncInput
14           _>true))))"
15          );
16  }
17
18  after():Main(){
19        AmoebaChecker.checker.terminate();
20  }
```

Figure 8.4: Instrumentation aspect definition for the `main()` method

the run-time model checking server through network communication. Figure 8.5 (b) shows the advice definition for SyncOutput. Line 1 defines that it is a *before advice* for the `sync_output` pointcut. Before each access to the sync buffer, the advice inserts an invocation of the `nextState()` method of the `checker` (Line 2), which sends the SyncOuput message to the run-time model checking server.

We executed the instrumented adaptive Java program and verified the program against the overlap adaptation requirement in Formula (8.3) using AMOEBA-RT. The model checker reported success in all the executions, indicating that the program satisfied the requirement in these executions.

In order to demonstrate that the model checker is actually effective in catching errors, in a second experiment, we partially removed the buffer empty condition check from the sync piped input. This time, AMOEBA-RT caught violations of the property in Formula 8.3 in some of the random executions. As a response to the violations, AMOEBA-RT recorded the execution paths in a bug report for future debugging purposes. The bug report in the above experiment showed that during those executions, the obligation fulfillment requirement was violated.

```
1 public pointcut sync_output():
2   withincode(*sync.PipedInputStream.receive(..))&&
3   get (byte[] buffer ) ;
```

(a) Pointcut definition for SyncOutput

```
1 before(): sync_output(){
2   checker.nextState("SyncOutput");
3 }
```

(b) Advice definition for SyncOutput

Figure 8.5: Instrumentation aspect definition for SyncOutput

8.3 Related Work

A limited number of run-time model checking projects have been investigated, including Java PathExplorer (JPaX) [55], the run-time monitoring and checking (MaC) architecture [83], temporal rover tools [35], EAGLE [9], etc. For example, JPaX [55] is a run-time model checker for Java programs. It supports temporal logic-based verification, including future time and past time linear temporal logics. The conformance between an execution and a logic expression is performed by a *Maude* rewrite engine. JPaX run-time model checking comprises three main modules: an instrumentation module, an observer module, and an interconnection module. The instrumentation module performs instrumentation of Java bytecode using an instrumentation script woven in by a static instrumentation tool, Jtrek. At run time, the instrumented Java programs emit events (pertaining to their run-time conditions) to the interconnection module. The interconnection module ties the instrumentation module and the observer module together by further transmitting the events to the observer module. The observer module, which may be running on a different machine, verifies the received events against a temporal logic specification. Our AMOEBA-RT is inspired

by JPaX. In the AMOEBA-RT architecture, the instrumentation module and the run-time model checking server module correspond to their instrumentation module and observer module, respectively. In our current implementation, we do not have the interconnection module, since we did not find it necessary for our proof-of-concept purpose so far. In the future, the interconnection module may be helpful in AMOEBA-RT in order to further decouple the instrumentation module from the model checking server module. The major difference between AMOEBA-RT and existing run-time model checking approaches is its support for model checking adaptation requirements specified in A-LTL.

Run-time model checking has also been applied to monitor executions of adaptive software. As described in Section 4.3, Feather *et al* [41] proposed using FLEA [31] to specify and monitor run-time properties in adaptive software. Again, as described in Section 4.3, the properties they specified and verified were all non-adaptive properties, since the FLEA temporal logic is not specifically designed for specifying adaptation behavior. Our approach extends theirs by adding the capability of specifying and monitoring adaptation behavior specified in A-LTL.

8.4 Discussion

In the software industry, testing is still the prevailing approach for software quality assurance. However, in large scale, complex software, the execution paths that can be covered by pre-release testing constitute only a small portion of all possible paths. Static model checking promises exhaustive coverage of all execution paths. Unfortunately, it suffers from the notorious state explosion problem. Run-time model checking provides a means to continuously monitor and verify the post-release behavior of a software system. When an error is detected, the software may either file a bug report, or try to repair the error automatically.

In this chapter, we introduced AMOEBA-RT, a run-time verification scheme for adaptive software. AMOEBA-RT includes two parts, an instrumentation module and a run-time model checking server. We used an aspect-oriented technique to instrument adaptive Java programs in order to separate the run-time model checking concern from the program. The AMOEBA-RT model checking server interprets A-LTL specifications and verifies execution sequences against A-LTL specifications at run time. We have applied AMOEBA-RT to the verification of the adaptive Java pipeline program and detected errors in our experiments. AMOEBA-RT currently supports filing bug reports. More sophisticated responses to run-time errors, such as self-healing and self-repairing, require feedback from the run-time model checking server to the adaptive program, which we plan to investigate in our future work.

There are several limitations that must be considered when using run-time model checking techniques. We next overview these limitations.

Performance overhead. Run-time verification occurs at run time. The overhead incurred by the instrumentation code includes two parts: (1) the overhead to initialize the model checking server at the beginning of each execution, and (2) the overhead to update state condition information on the model checking server during the execution. The first part includes sending an A-LTL formula to the model checking server and constructing the property automaton on the model checking server. The complexity of the automaton construction is exponential to the length of the formula. However, this overhead is encountered only once for every execution. Moreover, the automaton construction occurs on the server, which could be running on a different computer. Therefore, the impact of the overhead of this part to the run-time adaptive system performance is largely negligible. The overhead of the second part includes the evaluation of monitored state conditions and the transmission of these conditions

to the run-time model checking server. Its effect on the run-time adaptive program depends on the density of the instrumentation points, the number of conditions to be monitored at each point, and the encoding of the conditions to be transmitted. In our experiment with the adaptive Java pipeline example, the performance overhead incurred by run-time verification is largely imperceptible ($< 1\%$).

Non-exhaustive verification. Unlike static model checking, run-time model checking only checks one execution path at a time. Successes of run-time model checking increase our confidence in the correctness of the software, but they can never prove the correctness of the program in general.

After the fact repair. Since run-time model checking occurs at run time, when errors are detected, damages (e.g., leakage of sensitive information) may have already been made to the run-time system. Although the run-time model checking server could potentially invoke damage control routines to recover or repair the system, some types of damage caused by unforeseeable errors are irreversible, such as security leakage. Run-time verification is a useful complement to testing and static model checking rather than a replacement. Adaptive software must be thoroughly tested before release in order to reduce the number of errors at run time. If the software is to be executed in safety critical domains, then static model checking, as described in Chapter 7, must be performed to ensure critical properties in the system .

Chapter 9

Safe Dynamic Adaptation Protocol

This chapter describes a safe adaptation protocol for providing assurance to dynamically adaptive programs during program execution [150, 152]. This protocol ensures that adaptation steps are performed in a way that does not violate dependency relationships among components in distributed adaptive software. We assume that the requirements and design described in previous chapters have been refined in the implementation and have been described with communications and dependency relationships among components. The properties discussed in the chapter are much more fine grained and are dependent on run-time information. Our approach provides centralized management of adaptations, thereby enabling optimizations when more than one set of adaptive actions can be used to reach a target configuration. A rollback mechanism is provided if a failure is encountered during the adaptation process.

This chapter is organized as follows. In Section 9.1, we describe the theoretical foundation of our approach. Section 9.2 describes our proposed approach to safe adaptation in detail, and Section 9.3 describes its application in a video multicasting system. Related work is overviewed in Section 9.4, and Section 9.5 discusses limitations and extensions of the proposed approach.

9.1 Theoretical Foundations for Safe Adaptation

The adaptations that we consider in this chapter are component insertion, removal, replacement, and combinations thereof. A distributed component-based software system can be modeled as a set of communicating components running on one or more processes [77]. Components are considered to be communicating as long as there is some type of interaction, such as message exchange, function calls, network communication, and so on. A *communication channel* is the facility for communication, such as a TCP connection, an interface, etc. Communication channels are directed. A two-way communication between two components is represented with two channels with traffic traversing in opposite directions. A component can communicate with another as long as there exists a path of one or more channels connecting these two components.

In general, communication among components can be decomposed into multiple non-overlapping communication segments of various granularity. A coarse-grained segment can be divided into multiple finer-grained segments. For example, the communication between a video server and a video client can be divided into multiple transmit/receive sessions; each session can be divided into multiple frames, where each frame can be divided into multiple packets.

Communication can be either local or global. *Local communication* involves components of only one process, such as a local procedure. *Global communication* involves components from more than one process. A UDP datagram transmission over a network is an example of global communication, involving both a sender and a receiver processes.

Dynamic adaptations may interrupt ongoing communication. The communication whose interruption may cause errors in the system is termed a *critical communication segment*. We use a set of finite sequence of indivisible actions (named *atomic actions*) to model the set of critical communication segments *CCS*. The communi-

cation among components in a system is modeled as a (finite or infinite) sequence of (*critical communication identifier*, atomic action) pairs. Given a communication sequence, S, and a critical communication identifier, CID, we can extract from S the sequence of atomic actions with the same CID, denoted as S_{CID}. Given a set of critical communication segments CCS, we say that an adaptive system does not interrupt the critical communication segments if the communication sequence of the adaptive system is S and for all critical communication CID, we have $S_{CID} \in CCS$. Unsafe adaptation occurs when adaptive actions involving communicating components disrupt normal functional communication between the adapted component and the rest of the system, thus introducing system inconsistencies.

The formal requirements specifications introduced in Chapters 3 and 4 can be used to identify critical communication segments. For example, the liveness property $\Box(send \Rightarrow \Diamond ack)$ in the requirements specification implies that a "send" and "ack" action pair forms a critical communication segment, and therefore, adaptation may occur in the states after the acknowledgements for all the send operations have been received.

In a given system, multiple components may collaborate by communicating with each other. We use *dependency relationships* to model these communication patterns. The correct functionality of a component, c, may require the correct functionality of other component(s). The absence of other components may disrupt normal functionality of c. Based on the discussion above, we define a *safe dynamic adaptation process* as follows:

Definition: A dynamic adaptation process is ***safe*** if and only if:

- It does not violate dependency relationships among components.
- It does not interrupt critical communication segments.

In the following, we present our safe dynamic adaptation process.

9.1.1 Dependency Relationships

In a given system, if the correct functionality of a component A depends on a condition *Cond* to be true, then we say A *depends* on the condition, denoted as $A \Rightarrow Cond$, where "$\Rightarrow$" denotes a dependency relationship. The condition takes the form of a logic expression across the components. For example, $A \Rightarrow (B_1 \oplus B_2) \wedge C$ means that the correct functionality of component A requires the correct functionality of either component B_1 or B_2, and C, where the operator "$\oplus$" represents the logical "xor" operation, and "$\wedge$" represents the logical "and" operation. We use a special type of dependency relationship, *structural invariant*, to specify correct conditions of the system structure. These structural invariants specifiy the software structure in the implementation, which can be considered as a refinement of the global invariants described in Part I. For example, the structural invariant $A \wedge B$ indicates that the correctness of the system (i.e., to satisfy the global invariants described in Part I) depends on the correct functionality of both component A and component B.

In a safe adaptation process, the dependency condition of a component should always be satisfied when the component is in its fully operational state. Since dependency relationships are based on communication, if we block the communication channels of a component, then we may temporarily relax the dependency relationships and perform necessary adaptive actions. Before the communication in these channels is resumed, the dependency relationships must be reinforced.

Safe Configurations and Safe Adaptation Paths. A *system configuration* comprises a set of components that work together to provide services. If a dependency relationship predicate *dr* is evaluated to be true when we associate *true* to all components in a configuration, and associate *false* to all components not in the configuration,

then we say the configuration satisfies the dependency relationship. If a configuration satisfies all the dependency relationships, then this configuration is considered to be *safe*, otherwise, it is *unsafe*. A system can operate correctly only when it is in one of its safe configurations. All safe configurations can be obtained from the dependency relationships and available components.

A system moves from one configuration to another by performing *adaptive actions*. An adaptive action is defined as a function from one configuration to another: $adapt(config_1) = config_2$, where $config_2$ is the resulting system configuration when the adaptive action, *adapt*, is applied to $config_1$.

A *distributed adaptive action* comprises multiple local adaptive actions of individual processes. Each local adaptive action is divided into three parts: the *pre-action*, the *in-action*, and the *post-action*. The pre-action is the preparation operation, such as initializing new components, etc. The in-action alters the structure of the program. The post-action specifies tasks to be performed after the in-action, such as the destruction of old components. The pre-actions and post-actions do not interfere with the functional behavior of the adapting process.

We assume that an adaptive action is *atomic* and *isolated*. *Atomicity* of an adaptive action implies that the adaptive action should either not start or run to completion. *Isolation* of an adaptive action implies that the adaptive action is performed without interleaving with other operations.

An *adaptation step* is an ordered configuration pair: $step = (config_1, config_2)$, where *step* represents a system configuration transition from $config_1$ to $config_2$. A safe adaptation process comprises a set of safe configurations connected by a set of adaptation steps. These configurations and adaptation steps, together, form a *safe adaptation path* that starts from the source configuration of the first step and ends at the target configuration of the last step.

We can construct a *safe adaptation graph* (SAG) for an adaptive system, where

vertices represent safe configurations and arcs represent adaptation steps connecting safe configurations. A SAG can be constructed from a list of available adaptive actions. An adaptation step, ($config_1$,$config_2$), is in the SAG if and only if:

- Both $config_1$ and $config_2$ are safe configurations.
- There exists an adaptive action *adapt* such that $adapt(config_1)=config_2$.

9.1.2 Critical Communication Segments

Performing adaptive actions may disrupt communication among components. A safe adaptation process should maintain the integrity of critical communication segments. The system state in which an adaptive action does not interrupt any critical communication segments is called a *global safe state* for the action.

If a communication is local, then the integrity of its segments can be maintained by a local process. A local process is said to be in a *local safe state* for an adaptive action, if the action does not interrupt local critical communication segments. The integrity of global critical communication segments is guaranteed by a *global safe condition*, meaning that the adaptive action does not interrupt global critical communication segments. For example, the global safe condition for a UDP-datagram transmission is that the receiver has received all the datagram packets that the sender has sent, where the transmission of each datagram packet is a critical communication segment.

A system is in its global safe state if and only if:

- All the processes are in their local safe states.
- The global safe condition is satisfied.

9.1.3 Enabling Safe Adaptation

Next, we introduce the theoretical foundation for our safe adaptation process, and prove the process is safe.

Theorem 13: *The following two statements are equivalent.*

(a) An adaptation process is safe.

(b) The adaptation process is a process that executes along a safe adaptation path, where each adaptive action along the path is performed in its global safe state.

Proof 13: *Proof sketch:*

1. $(b) \rightarrow (a)$.

   If an adaptation process is performed along a safe adaptation path and each adaptive action is performed in a global safe state, then during the adaptation process, the system is either at a safe configuration or in a transition from one safe configuration to another.

   When the system is at a safe configuration, it does not violate dependency relationships (definition of safe configuration). Because no adaptive action is performed, critical communication segments will not be interrupted due to adaptation.

   Adaptive actions are performed in global safe states, which implies that no critical communication segments will be interrupted. Since adaptations start and end in safe configurations, dependency relationships will not be violated before and after the adaptive action. Adaptive actions are atomic, and thus we can assume there is no intermediate state during an adaptive action. Therefore, dependency relationships are not violated during adaptive actions.

2. *Use proof by contradiction to establish* $(a) \rightarrow (b)$.

   If (b) does not hold, then there are two possibilities: (1) the process is not performed along a safe adaptation path or (2) there is an adaptive action taking place in a state that is not globally safe. In the first situation, there must be a configuration on the adaptation path violating dependency relationships, and therefore, the adaptation process is unsafe. In the second situation, the adaptive action might interrupt a critical communication segment, and thus, the adaptation process is unsafe. Therefore, if (b) does not hold, (a) cannot hold.

9.2 Safe Adaptation Process

The safe adaptation is executed by an *adaptation manager*, typically a separate process that is responsible for managing adaptations for the entire system. The adaptation manager communicates with *adaptation agents* attached to processes involved in the adaptation. An agent receives adaptive commands from the adaptation manager, performs adaptive actions, and reports the status of the local process to the adaptation manager. Communication channels can be implemented to best match the communication patterns of the particular system. For example, both Arora [7] and Kulkarni [77] have used spanning trees, which are well suited to components organized hierarchically. In contrast, in a group communication system, multicast may be a better mechanism for the coordination between the adaptation manager and the agent processes.

Our approach comprises three phases: the analysis phase, the detection and setup phase, and the realization phase. The analysis phase occurs during development time. In this phase, the programmers should prepare necessary information such

as determining dependency invariants, specifying critical communication segments, etc. The detection and setup phase occurs at run time. When the system detects a condition warranting an adaptation, the adaptation manager should generate a safe adaptation path. In the realization phase, the adaptation manager and the agents coordinate at run time to achieve the adaptation along the safe adaptation path established during the previous phase.

9.2.1 Analysis Phase

At development time, the adaptive software developers should prepare a data structure $P = (S, I, T, R, A)$, where

- S is the set of all configurations.
- I (I: $S \rightarrow BOOL$) is the conjunction of the set of dependency relationship predicates evaluated upon configurations. $I(s)$ is *true* if and only if the system satisfies all the dependency relationships in the configuration s.
- T is a set of adaptive actions.
- R (R: $T \rightarrow PROGRAM$) maps each adaptive action to its corresponding implementation code in the program, where $PROGRAM$ represents the implementation. The reconfiguration is achieved by the execution of the implementation code.
- A (A: $T \rightarrow VALUE$) is a cost function that maps each adaptive action in T to a cost value in $VALUE$. We associate a fixed cost to each adaptive action. Factors affecting cost values include system blocking time, adaptation duration, delay of packet delivery, resource usage, etc.

9.2.2 Detection and Setup Phase

Once the system detects a condition warranting adaptation, the adaptation manager obtains the target configuration and prepares for the adaptation. This phase contains three steps.

1. **Construct Safe Configuration Set.** Based on the source/target configurations of an adaptation request and dependency relationships, this step produces a set of safe configurations.

2. **Construct Safe Adaptation Graph.** Next, we construct a safe adaptation graph (SAG) that depicts safe configurations as nodes and adaptation steps as edges.

3. **Find Minimum Safe Adaptation Path (MAP).** Finally, we apply Dijkstra's shortest path algorithm to the SAG to find a feasible solution with minimum weight, where the weight of a path is the sum of the costs of all the edges along the path.

9.2.3 Realization Phase

This phase requires the coordination of the adaptation manager and the agents at run time to carry out the actual adaptation according to the safe adaptation path. The adaptation manager should ensure that each adaptive action is performed in its global safe state. We use state diagrams to describe the behavior of each agent and the adaptation manager, respectively.

The state diagram of an agent at each local process is shown in Figure 9.1, where the `Courier` font denotes message names. Before an adaptive action is performed, each agent is in a **running** state. In this state, every component in the process is running in its full operation. When the agent receives a `reset` message, it moves

to a **resetting** state. The agent performs the local pre-action, and initiates a reset of the process. In the **resetting** state, the process is only partially operating: Some functionalities related to the adapted component are disabled. When the process reaches its local safe state, the agent performs a hold action to hold the process in the safe state, so that the local in-action can be performed safely. Then the agent sends the adaptation manager a `reset done` message, after which the process is in a **safe** state. In this state, the agent will perform its local in-action. When the in-action has finished, the agent sends the adaptation manager an `adapt done` message and reaches an **adapted** state. If the process is the only one involved in the adaptive action, then it can directly proceed to a **resuming** state from the **adapted** state without blocking. Otherwise, the process needs to remain blocked in the **adapted** state until it receives a `resume` message. When the agent receives a `resume` message, it knows that all processes have completed their adaptive in-actions, and thus proceeds to a **resuming** state, where the agent attempts to resume the full operation of the process. Finally, when the full operation of the process is resumed, the agent sends the manager a `resume done` message and performs the local post-action of the adaptive action, returning to the **running** state.

The state diagram of the adaptation manager is shown in Figure 9.2. The adaptation manager starts from a **running** state where the system is fully operational. When an adaptation request is received by the adaptation manager and a MAP is created after the planning phase, it sends `reset` messages to all the agents. Sending the first `reset` message brings the adaptation manager to an **adapting** state. In this state, the adaptation manager waits for the `adapt done` messages from all agents. When all `adapt done` messages are collected, the adaptation manager proceeds to an **adapted** state. Then the adaptation manager sends `resume` messages to the agents and the manager proceeds to the **resuming** state. When the adaptation manager collects `resume done` messages from all agents, it transitions to the **resumed** state. If

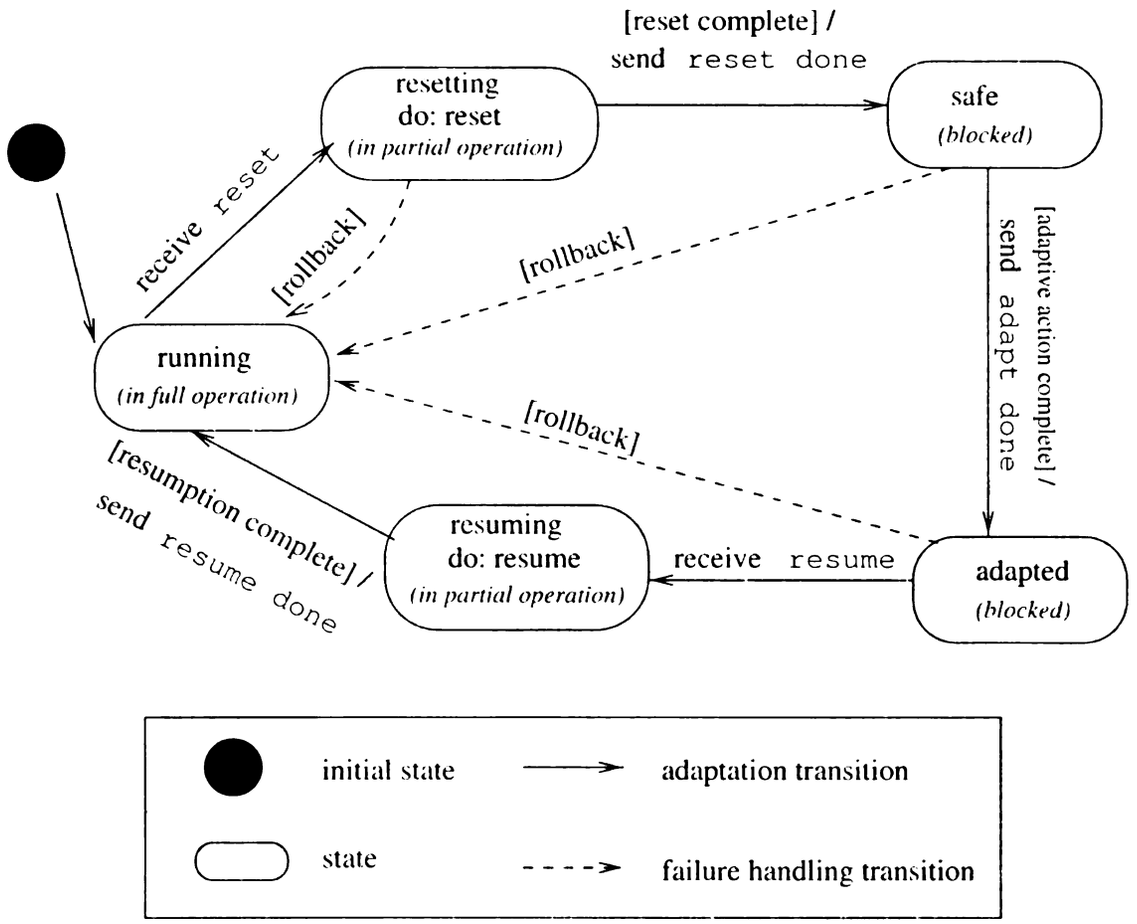


Figure 9.1: State diagram of a local process during adaptation

there are more adaptation steps remaining in the adaptation path, then the adaptation manager will repeat the traversal of **preparing**, **adapting**, **adapted**, **resuming**, and **resumed** states until the system configuration matches the target configuration. When the last adaptation step has finished, the adaptation manager returns to the **running** state.

9.2.4 Failures During Adaptation Process

We identify two major types of failures based on our experience. First, if the communication between the manager and the agents is unreliable, then the messages between them may be lost, causing *loss-of-message* failures. Second, when the agent of a local process receives a `reset` message, the local process may not be able to reach

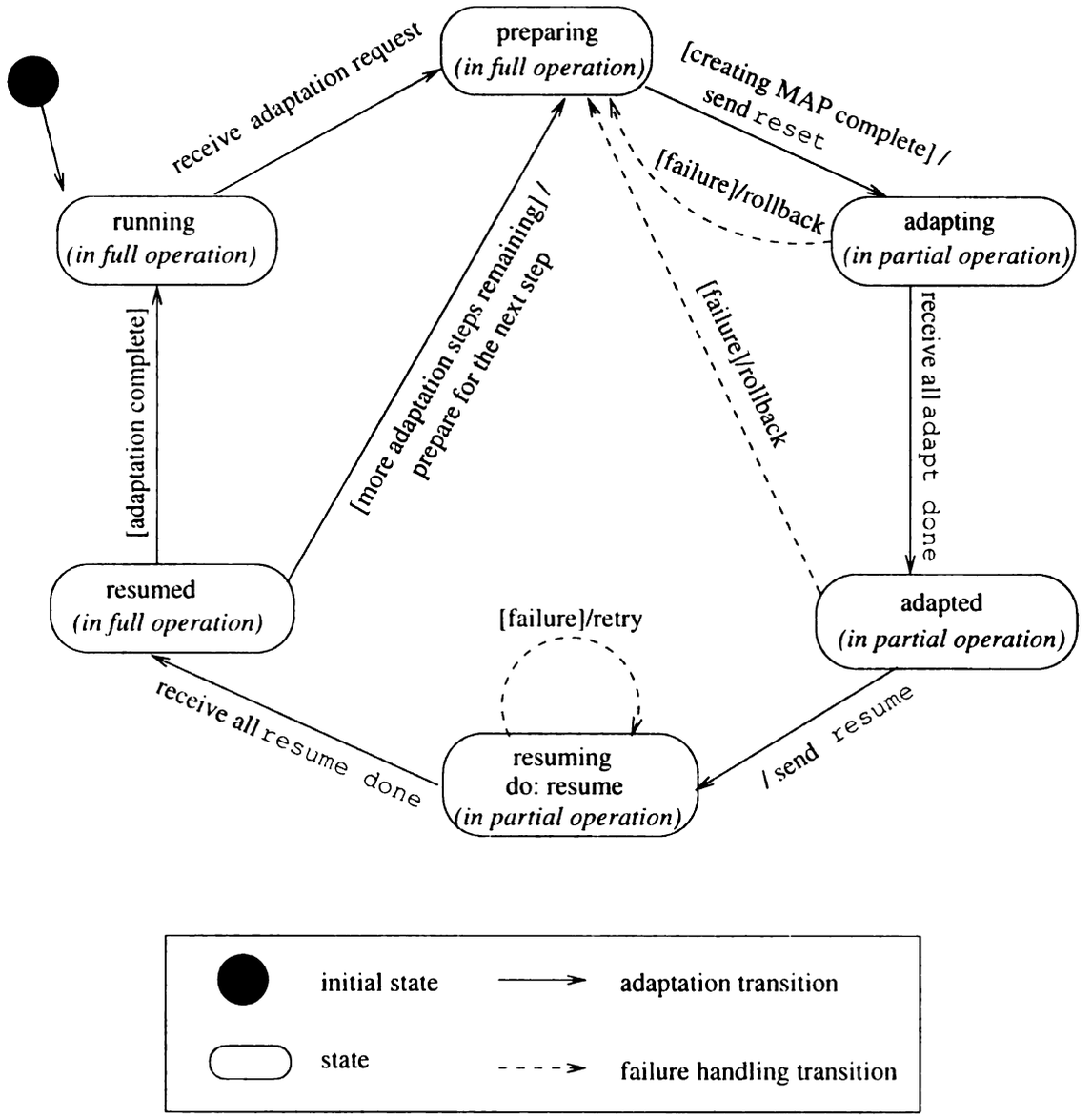


Figure 9.2: State diagram of the adaptation manager during adaptation

a safe state in a reasonably short period of time, thus causing a *fail-to-reset* failure. Both types of failures can be detected by a time-out mechanism on the manager.

Loss-of-Message Failure. Loss-of-message failures caused by transient network failures can be handled by multiple attempts to send the messages. However, loss-of-message failures caused by permanent network failures may cause system inconsistencies if the system does not respond to this type of failures correctly. The general rule for handling loss-of-message failures is that if the failures occur before the manager

sends out the first `resume` message, then the adaptation should be aborted. That is, the manager should stop sending any new `reset` and `adapt` messages and all the participating processes should roll back to the state prior to the adaptation. If the failures occur after the manager has sent out a `resume` message, then the adaptation should run to completion. That is, all the participating processes should eventually complete adaptation and resume.

Fail-to-Reset Failure. In some cases, when an agent receives a `reset` message, the local process may be engaged in a long critical communication segment, which may prevent it from reaching a safe state in a reasonably short period of time, thus causing a fail-to-reset failure. If a process cannot reach a safe state after it has received a `reset` message, then the adaptation process should be aborted, and all affected processes should roll back to the state prior to the adaptation.

Failure Handling Strategies. In the event that a failure occurs during an adaptation step, there are two possible outcomes: (1) The adaptation step succeeds and the system reaches the target safe configuration. (2) The adaptation step fails and the system reaches a safe configuration prior to the adaptation. If the adaptation step succeeds, then the manager should continue processing the remaining adaptation steps if there are any. If the adaptation step fails, then the manager has four options: (1) Retry the same step. (2) Try other adaptation paths. (3) Attempt to return to the source configuration. (4) Remain in the current safe configuration and wait for user intervention. We suggest using the combination of all options: The adaptation manager first tries the same step a second time. If it fails again, then it tries an alternative adaptation path from the current configuration to the target configuration. If all possible paths to the target configuration have been tried and have failed, then the adaptation manager tries to return to the source configuration. If this attempt also fails, then the adaptation manager notifies the users and waits

for user intervention.

The dashed arrows in Figures 9.1 and 9.2 show the failure handling transitions on both the manager and the agents. We claim that the adaptation process is still safe with the presence of failures. During an adaptation step, a rollback is invoked only when no process has been resumed, which ensures that no side effect is produced before the rollback. Otherwise, the adaptation will run to completion, which has the same effect as if the adaptation had no failures.

9.3 Case Study: Video Streaming

We use a video multicasting system to illustrate the safe adaptation process. The system setup is similar to the audio streaming example introduced in Chapter 5. However, in this section, we use different configurations on the clients. Figure 9.3 shows the initial configuration of the application, comprising a video server and one or more video clients. In this example, one client is a hand-held computer (e.g., iPAQ) with a short battery life and limited computing power, and the second client is a laptop computer (e.g., Toughbook) with reasonable computing power, but limited battery capacity. On the server, a web camera captures video input and a video processor encodes the stream. The encoded video, already packetized, is delivered to the network through a MetaSocket. After traversing a chain of zero or more (encoder) filters, the packets are eventually transmitted on a multicast socket. On each client, the packets are processed by a chain of decoder filters in a receiving MetaSocket. Subsequently, they are passed to the video processor, where they are unpacketized into video frames. Finally the frames are displayed on video players.

In this example, two main encryption schemes are available for processing the data: DES 64-bit encoding/decoding, and DES 128-bit encoding/decoding. The sender has two components: E1, a DES 64-bit encoder and E2, a DES 128-bit en-

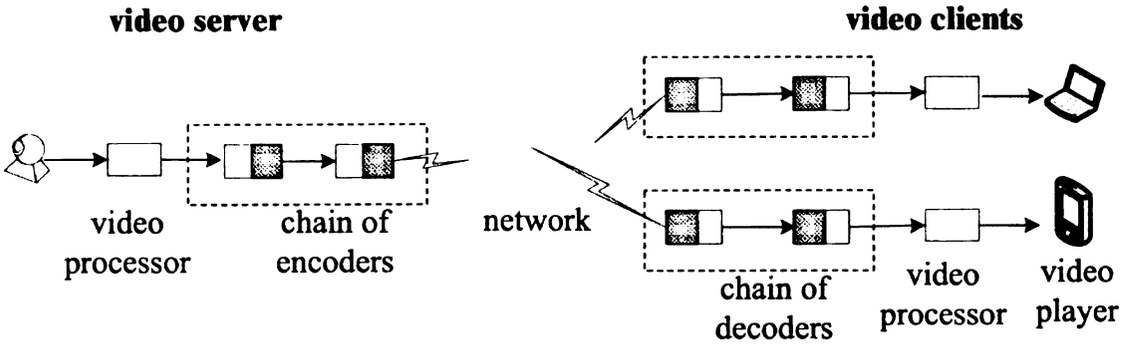


Figure 9.3: Configuration of the video streaming application

coder. The hand-held client has three components: D1, a DES 64-bit decoder, D2, a DES 128/64-bit compatible decoder, and D3, a DES 128-bit decoder. The laptop client has two components: D4, a DES 64-bit decoder and D5, a DES 128-bit decoder. In general, a DES encoder generates DES encrypted packets from plain packets and a DES decoder decrypts the DES encrypted packets. Each decoder implements the "bypass" functionality: When it receives a packet not encoded by its corresponding encoder, it simply forwards the packet to the next filter in the chain. The available adaptive actions are: (1) inserting, removing, and replacing a single encoder or decoder; (2) inserting, removing, and replacing an encoder/decoder pair; (3) inserting, removing, and replacing an encoder/decoder triple. The overall adaptation objective is to reconfigure the system from running the DES 64-bit encoder/decoders to running the DES 128-bit encoder/decoders to "harden" security at run time. We use a separate process to implement the adaptation manager and attach an agent thread to both the server and the clients, respectively.

9.3.1 Safe Adaptation Path

By analyzing the communication patterns between the encoders and the decoders, we find that the correct functionality of a decoder does not require an encoder, but in order to decode a packet generated by an encoder, there must be a

corresponding decoder for each encoder. We have the following invariants, where $\bigoplus$ represents "exclusively select one from a given set of elements".

- System Invariants:
  - **Resource constraint:** $\bigoplus(D1, D2, D3)$.

    The receivers allow only one DES decoder instance in each device at any given moment due to computing power constraints.
  - **Security constraint:** $\bigoplus(E1, E2)$.

    The sender should have one encoder in the system so that all data are encoded during adaptation.
- Dependency invariants:
  - $E1 \Rightarrow (D1 \vee D2) \wedge D4$.

    $E1$ encoder requires the $D1$ or $D2$ decoder to work with the $D4$ decoder.
  - $E2 \rightarrow (D3 \vee D2) \wedge D5$.

    $E2$ encoder requires the $D3$ or $D2$ decoder to work with the $D5$ decoder.

We input source and target configurations and the above dependency invariants to the adaptation manager, which generates the safe configuration set. For brevity and automatic processing purposes, we use a 7-bit vector **(D5, D4, D3, D2, D1, E2, E1)** to represent a configuration: If the corresponding bit is "1", then the component is in the configuration, otherwise, it is not in the configuration. The source configuration is (0100101) and the target configuration is (1010010).

The resulting safe configuration set is shown in Table 9.1. The adaptive actions shown in Table 9.2 are input to the adaptation manager. Only relevant actions are listed. The cost column gives packet delay in milliseconds. Note that in order to perform some of the actions (e.g., A6-A9), the server has to be blocked until the last

packet processed by the encoder has been decoded by the decoder(s) on the client(s). As a result, these actions are much more expensive than other actions.

bit vector	configuration	bit vector	configuration
0100101	D4,D1,E1	1100101	D5,D4,D1,E1
1101001	D5,D4,D2,E1	1101010	D5,D4,D2,E2
1110010	D5,D4,D3,E2	0101001	D4,D2,E1
1001010	D5,D2,E2	1010010	D5,D3,E2

Table 9.1: Safe configuration set

Action	Operation	Cost (ms)	Description
A1	$E1 \rightarrow E2$	10	replace E1 with E2
A2	$D1 \rightarrow D2$	10	replace D1 with D2
A3	$D1 \rightarrow D3$	10	replace D1 with D3
A4	$D2 \rightarrow D3$	10	replace D2 with D3
A5	$D4 \rightarrow D5$	10	replace D4 with D5
A6	$(D1, E1) \rightarrow (D2, E2)$	100	A1 and A2
A7	$(D1, E1) \rightarrow (D3, E2)$	100	A1 and A3
A8	$(D2, E1) \rightarrow (D3, E2)$	100	A1 and A4
A9	$(D4, E1) \rightarrow (D5, E2)$	100	A1 and A5
A10	$(D1, D4) \rightarrow (D2, D5)$	50	A2 and A5
A11	$(D1, D4) \rightarrow (D3, D5)$	50	A3 and A5
A12	$(D2, D4) \rightarrow (D3, D5)$	50	A4 and A5
A13	$(D1, D4, E1) \rightarrow (D2, D5, E2)$	150	A1 and A10
A14	$(D1, D4, E1) \rightarrow (D3, D5, E2)$	150	A1 and A11
A15	$(D2, D4, E1) \rightarrow (D3, D5, E2)$	150	A1 and A12
A16	$-D4$	10	remove D4
A17	$+D5$	10	insert D5

Table 9.2: Adaptive actions and corresponding cost.

The adaptation manager creates the SAG shown in Figure 9.4 and uses Dijkstra's shortest path algorithm to obtain the shortest path (A2, A17, A1, A16, A4), which in this example, has cost 50 ms.

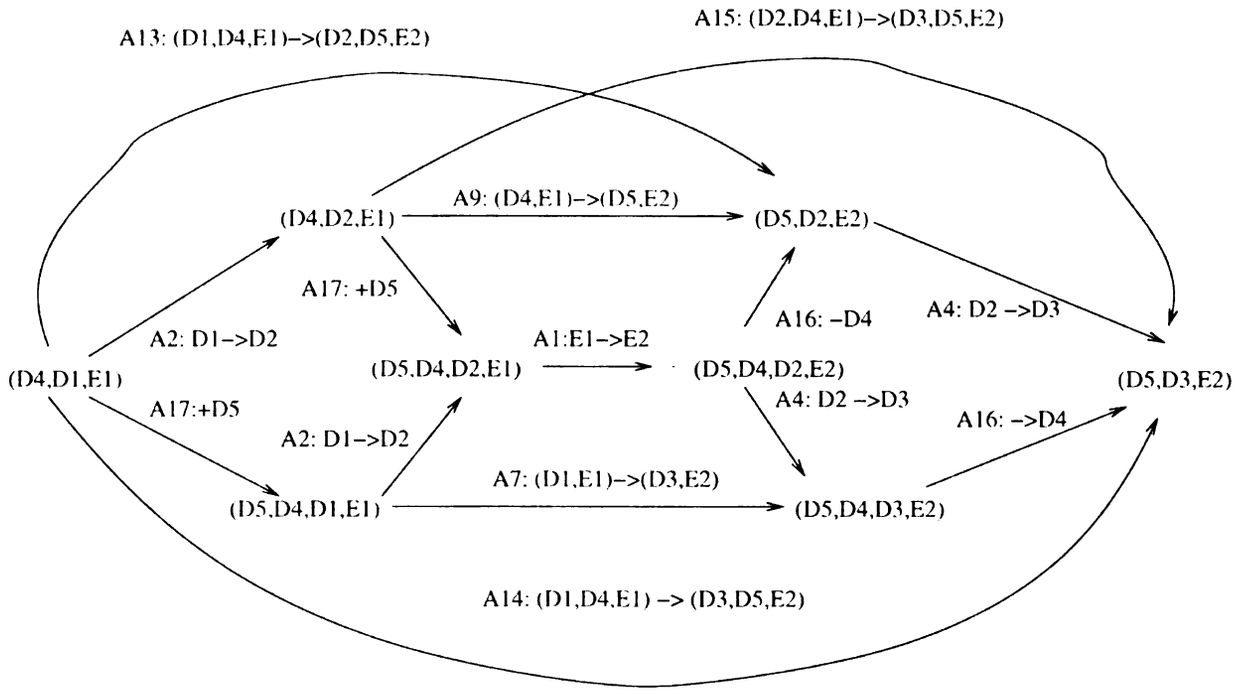


Figure 9.4: Safe adaptation graph (SAG)

9.3.2 Performing Adaptive Actions Safely

The adaptation steps for the safe adaptation path are:

Step (1). **Action A2:** Replace D1 with D2.

Step (2). **Action A17:** Insert D5.

Step (3). **Action A1:** Replace E1 with E2.

Step (4). **Action A16:** Remove D4.

Step (5). **Action A4:** Replace D2 with D3.

Action $A2$ in step (1) only involves the process running the MetaSocket on the hand-held device. The adaptation manager sends a `reset` message to the agent on the hand-held device. The global safe state of this action is the same as the local safe state of the device: the DES decoder is not decoding a packet. When the agent receives the `reset` message, it sets a "resetting" flag in the MetaSocket. When the

decoder finishes decoding a packet, it checks the "resetting" flag. If it is set, then it notifies the agent and blocks itself. The agent sends a `reset done` message to the adaptation manager and performs the $(A2 : D1 \rightarrow D2)$ action. When the adaptive action is done, it sends an `adapt done` message to the adaptation manager. The agent directly resumes the hand-held's full operation and sends a `resume done` message to the manager. Other steps (2-5) can be performed in a manner similar to that used in step (1).

9.4 Related Work

Numerous techniques have been proposed to address dynamic adaptation at run time. Kulkarni *et al* [77] proposed a distributed approach to safely adapt (i.e., component replacement) distributed fault-tolerance components at run time. In their work, a *distributed component* is defined as a composition of a number of *component fractions*, each of which is installed at a different process in a system. There exist dependency relationships among these component fractions, and thus certain components need to be blocked before they can be safely removed from the system. Their goal is to achieve system wide substitution of all the component fractions of a distributed component so that the dependency relationships are not violated and the blocking of the component fractions is minimized. Towards this end, they proposed a *distributed reset* algorithm, which works in two steps: In the first step, termed *initialization wave*, initialization messages are sent from a designated process to its neighbors (directly connected processes). Upon receiving the initialization message, a process first relays the message to its own neighbors, and then resets its state to prepare for the component replacement. At the end of the initialization wave, the paths of the initialization messages form a spanning tree. In the second step, termed *replacement wave*, replacement messages are passed along the spanning tree to all

the processes in the tree in order to replace component fractions running on these processes. Using the above distributed reset, they achieved distributed component replacement with local minimal blocking (i.e., a component fraction is blocked only when it is replaced). The safe adaptation protocol introduced in this chapter is inspired by their approach. However, we used a centralized management to minimize the cost for adaptation. The cost in our approach includes blocking time, resource usage, and other factors. Also, the minimum cost achieved by our approach is global, i.e., among all possible safe adaption paths, while their minimal blocking is local to a given adaptation path.

Chen *et al* introduced Cactus [22], a system for constructing highly configurable distributed services and protocols. In Cactus, a host is organized hierarchically into layers, where each layer comprises a number of adaptive components (ACs). Each AC contains multiple alternative adaptation-aware algorithm modules (AAMs) providing alternative implementations of a service. During normal operation (i.e., not adapting), only one AAM is running in each AC. The ACs in each layer receive *incoming messages* from the layer below it, process the messages, and then pop them up to the layer above it. Similarly, they receive *outgoing messages* from the layer above it, process the messages, and then push them down to the layer below it. They proposed a *graceful adaptation protocol* to coordinate the adaptation of adaptive components (switching from running AAM_{old} to running AAM_{new}) across multiple hosts. The protocol comprises three key steps: *preparation*, *outgoing switchover*, and *incoming switchover*. During preparation, each AC prepares to receive and buffer incoming messages produced by other AAM_{new}. During the outgoing switchover, each AC that is responsible for outgoing messages switches from running AAM_{old} to running AAM_{new}. In incoming switchover, triggered by receiving an incoming message generated by AAM_{new}, each AC responsible for incoming messages switches from running AAM_{old} to running AAM_{new}. The graceful adaptation protocol avoids inflight message-loss

during adaptation and reduces message blocking incurred by the adaptation. Their approach implicitly assumes a simple dependency rule among ACs, i.e., the senders depend on the receivers. However, in some systems, the dependency relations may be more complex than that, such as in the case introduced in this chapter. Therefore, their approach cannot be directly applied to address the safe adaptation problem in this chapter.

Appavoo *et al* [5] proposed a *hot-swapping* technique that supports run-time object replacement in an open source research operating system, *K42*, developed at IBM research. In K42, every service is encapsulated in a component, represented by an object. The services of the operating system can be adapted by replacing old objects with new ones. The key infrastructure supports in K42 for hot-swapping include: (1) The references of the objects are made indirectly through an *Object Translation Table* (OTT). (2) A generation count scheme is used to determine whether the object has reached a state in which the object is not being used by other objects. (3) Object state transfer negotiation protocols are used to transfer state information from an old object to a new object. The hot swapping process of an object includes three stages. In the first stage, the OTT entry of the object points to the old object. In the second stage, the OTT entry points to a mediator, and the mediator performs the hot-swapping from the old object to the new object. In the third stage, the OTT entry points to the new object. The second stage includes the following phases.

1. **Preparation phase**: In the preparation phase, the system inserts a mediator between the OTT and the object. The mediator forwards calls to the original object, and begins to track new calls to the object.

2. **Blocking phase:** When all the calls initiated before the preparation phase have completed, the adaptation goes to the blocking phase. In the blocking phase, new calls to this object are blocked until all the calls tracked by the mediator have completed.

3. **Quiescence phase**: When all the calls initiated in the preparation phase have finished, the adaptation is in the quiescence phase. In the quiescence phase, the system performs state transformation from the old object to the new one.

4. **Complete phase**: After the state transformation is done, the adaptation enters the complete phase. In this phase, the mediator is removed, the OTT entries point to the new object, and blocked calls are delivered to the new object.

The hot-swapping technique is specifically designed for the K42 operating system. Their approach assumes that all the calls can be completed in a short period of time, thus the object blocking is minimized. Their approach also assumes that there are no dependency relationships among the calls, since such kind of dependencies may cause deadlocks in the adaptation process with their approach, which we explain as follows. For example, if the completion of a call C_i initiated in the preparation phase depends on the completion of another call C_j initiated in the blocking phase, then C_i will never complete since C_j is blocked, and thus a deadlock is incurred. Therefore, the hot-swapping technique is not applicable to the safe adaptation problem introduced in this chapter.

9.5 Discussion

This chapter presented an approach to safe dynamic adaptation. We use a centralized adaptation manager to schedule the adaptation process, which results in a globally minimum solution. We block adapting components until the system has reached a new safe state and thus avoid unsafe adaptation. We also use timeout and rollback mechanisms to deal with possible failures during the adaptation process to ensure consistency in the system.

The interaction between the manager and the agents in our approach is similar to the two-phase commit protocol [53] if we combine the safe state with the adapted

state in the agents. However, in this work, we consider it clearer to have two separate states. Moreover, our protocol handles multiple adaptation steps where failures may occur at various phases of each step, whereas the two-phase commit protocol only addresses a single adaptation step.

Scalability is a concern for our technique. Because our technique searches the optimal path in a SAG, the computational complexity may be high when there are numerous adaptive components in the system (exponential to the number of components participating in an adaptation). To handle the complexity, we divide the adaptive components of a system into multiple collaboration sets where components collaborate with only those in the same set. The component adaptation of each set can be handled independently, thereby reducing the complexity

Chapter 10

Conclusions and Future Investigations

In this thesis, we introduced an approach to systematically apply formal methods to different phases in the development of adaptive software in order to gain assurance in adaptation. These techniques are summarized as follows:

(1) In the **requirements analysis** stage, we analyze and formally specify the requirements for adaptive software. We start from the high-level goals for the software and elicit the set of possible domains in which the software executes [144]. In order to systematically analyze the requirements for an adaptive software, we proposed the goal-based adaptation requirements analysis. We separately specify the requirements for each domain (local requirements), for adaptations from one domain to another (adaptation variant properties), and for the overall system (global requirements) [143, 146]. We proposed A-LTL [143, 146] to formally specify adaptation requirements and TA-LTL to formally specify real-time properties in adaptive software [153, 155].

(2) In the **model design and analysis** stage, we design and analyze formal design models from adaptive software. In the MASD approach [144], we first construct a steady-state model (e.g. Petri net) for each domain, then design adaptations

among steady-state programs using adaptation models. After the models are created, we perform model analyses, including simulation and model-checking, on the models to verify and validate the models against the requirements specified in the requirements stage. We also introduced a model-based technique to provide assurance in re-engineering legacy software for adaptation [149]. The adaptation design is performed upon UML Statechart models, which are initially extracted from legacy code using the metamodel-based language translation technique. We add adaptive states and transitions to these UML Statechart models to construct adaptation models. These models are automatically translated to formal specification languages, e.g., Promela, for formal analysis. We introduced the cascade adaptation technique and an aspect-oriented adaptation enabling technique to translate the UML models into adaptive implementations.

(3) In the **implementation** stage, we provide assurance in the adaptive software implementation with formal analysis and safe adaptation protocols. We use the models created from the design stage as blueprints to implement the adaptive software, and verify the implementations against requirements [144, 149]. We use the safe dynamic adaptation protocol to orchestrate adaptive actions among distributed adaptive components so that they take place in safe states, along safe adaptation paths, thus ensuring that the dependencies among these components are not violated during execution [143, 146]. In order to ensure that the implementation for adaptive software satisfies critical properties elicited in the requirements analysis stage, we developed a modular model checking technique AMOEBA that statically verifies adaptive software against A-LTL properties and significantly reduces the complexity of model checking [144]. For programs that are too complex to be verified statically, we also extended AMOEBA to be applied at run time to monitor and verify conditions in a running adaptive software.

10.1 Contributions

The following is a list of contributions made by the thesis.

Requirements Phase.

- We proposed a finite state machine model for adaptive software [144, 145]. This model enables existing analysis techniques for traditional, non-adaptive software to be directly applied to adaptive software. Also, it enables us to exploit specific characteristics of adaptive software in order to optimize their analysis [144] (Chapter 2).

- We introduced A-LTL, the adapt operator-extended linear temporal logic. We used A-LTL to formally specify adaptation properties of adaptive software [143, 146] (Chapter 3).

- We generalized three adaptation semantics, namely the one-point adaptation, the guided adaptation, and the overlap adaptation, and formally specified the semantics in A-LTL [143, 146] (Chapter 3).

- We introduced a goal-based technique to systematically analyze the requirements for adaptive software and to generate formal requirements specifications in A-LTL [144, 149] (Chapter 4).

- We introduced TA-LTL, a real-time extension to A-LTL for specifying critical real-time constraints in adaptive software. Three types of critical properties are identified, namely safeness, liveness, and stability properties [153, 155] (Appendix A).

Design Phase.

- We proposed a formal design modeling technique for adaptive software. We identified the key features of adaptive software design (i.e., quiescent states,

adaptive states, and adaptive transitions) and introduced a state-based modeling approach (MASD) to capture these features with design models [144]. We also introduced rapid prototyping and model-based testing to carry these features in the models to their implementations [144] (Chapter 5).

- We proposed a model-based re-engineering technique to enable adaptation in legacy software with assurance by leveraging the MASD approach [144], the metamodel-based language translation technique [97], and the aspect-oriented adaptation enabling technique [139]. We also introduced the cascade adaptation mechanism to handle state transformation from a source program to a target program in an adaptation [149] (Chapter 6).

Implementation Phase.

- We developed AMOEBA, a modular model checker that modularly verifies adaptive software against both LTL and A-LTL properties. The proposed technique reduces the model checking complexity by a factor of n, where n is the number of steady-state programs encompassed by the adaptive program [145] (Chapter 7).

- We developed a run-time model checker AMOEBA-RT that monitors the run-time conditions in adaptive software and verifies these conditions against LTL and A-LTL properties (Chapter 8).

- We designed a safe software adaptation protocol that minimizes the adaptation cost and ensures consistencies among adaptive components. A retry/roll-back mechanism is employed to ensure the consistency of adaptive actions in the presence of failures [150, 152] (Chapter 9).

10.2 Future Investigations

We now discuss a number of future research directions in order to provide assurance in dynamically adaptive software. The following areas are natural extensions to the work presented in the dissertation.

- **Integrated tool suites.** This thesis introduces a number of techniques to provide assurance in adaptive software. These techniques are intended to be applied in a coordinated manner. Many parts of our techniques are automated, with the remaining parts amenable to automation. Our approach leverages existing tools, including Renew, Maria, Hydra, Spin, Rational XDE, etc. We have also developed prototype tools to support our techniques, including AMOEBA and AMOEBA-RT. However, the tool support for our proposed approach can be further improved in the following respects: (1) New tools are required for certain tasks, including the extraction of adaptation state machines from source code [26, 28, 117], automatic selection of off-the-shelf components to be used as/in steady-state programs [85, 87, 125], automatic translation from adaptation models (e.g., UML Statechart models) to adaptive programs (e.g., Java) [97], and so on. (2) Existing tools are not sufficiently mature or easily integratable to be widely adopted. (3) Existing tools [78, 94] tend to use their own specific representations, making them difficult to be used together in practice, i.e., they are incompatible with each other. Based on the above observations, we propose building an integrated tool suite in order to promote practical applications of our proposed formal techniques to the development of adaptive software.

- **Integrate ADL and MASD.** The MASD approach introduced in Chapter 5 focuses on the behavioral aspect of adaptive programs. We also realize the importance of expressing adaptations at the architecture level, where structural changes of adaptive software are modeled as disconnections and reconnections

of connectors between components. We believe an integration of our approach with an appropriate ADL representation, including Darwin [92] and Wright [2], will provide a comprehensive solution to both structural and behavioral changes in the development of dynamically adaptive software.

- **Real-time adaptation model and model checking.** Although we have proposed TA-LTL as a real-time temporal logic for adaptive software, the support for real-time adaptive software is still limited. A formal model for real-time adaptive programs needs to be defined, and a model checking algorithm is also required. We hypothesize that these can be accomplished by extending our adaptive program model (in Chapter 2) and the model checking algorithm for A-LTL (in Chapter 5) with real-time constructs introduced in other real-time formal representations [3, 30, 37, 48, 88, 89].

- **Support for run-time changes of requirements.** This thesis focuses on the adaptation scenarios where the possible execution domains can be determined at development time, and the requirements for each domain are fixed at run time. In some autonomic computing systems, the software may need to adapt to new execution domains that are unknown to the developers before run time. The requirements for a domain may also be changed at run time to respond to unanticipated changes in the domain, or to achieve different goals. Techniques are required to address this type of requirement changes at run time.

- **Incorporate product line concept.** The re-engineering technique introduced in Chapter 6 requires a legacy database comprising different implementations to be used in various execution domains. These implementations could be in the form of a product line or program families [108]. It is worthwhile to study the common properties of these programs and to encapsulate the differences among these programs. Accordingly, the adaptation design and implementation

technique can be optimized.

- **Integrate AMOebA with modular verification of features.** In the modular verification technique introduced in Chapter 7, we did not place any constraints on the structure of the different steady-state programs. In practice, they usually share significant commonalities in their behavior, and therefore, may be constructed using compositions of features, where a feature is a piece of code pluggable to another piece of code (called *base*) at development time [86]. Fisler *et al* proposed a modular approach to verifying feature-oriented software [45]. It will be interesting to study a technique that leverages the modular feature verification technique used in AMOEBA so that not only adaptive software, but also each steady-state program itself can be verified modularly and incrementally. As a result, the complexity for model checking adaptive software can be further reduced.

10.3 Final Thoughts

As such, more rigorous software development techniques specially focusing on adaptation needs are needed. This dissertation presents the first collection of techniques that addresses assurance for adaptive systems, that starts with requirements and takes it all the way to implementation and testing, including re-engineering legacy systems for adaptation.

Appendix A

Timed A-LTL

In this chapter we introduce the *Timed* A-LTL (TA-LTL) [153, 155], an extension to A-LTL that includes a metric for time [37, 153]. Specifically, TA-LTL enables temporal constraints to be expressed in a quantitative form, in addition to specifying relative temporal ordering. We assume that the adaptation from one program to another has been specified with an A-LTL formula. TA-LTL enables us to specify three types of properties of adaptation that involve absolute time, namely the *safety*, *liveness*, and *stability* properties. The remainder of this chapter is organized as follows. Section A.1 provides background information on an underlying logic TPTL [3]. In Section A.2, we introduce the syntax and semantics of TA-LTL, and Section A.3 describes the use of TA-LTL in specifying the timing properties of the three commonly occurring adaptation semantics. Section A.4 illustrates the approach with a mobile communication example. Related work is overviewed in Section A.5. Section A.6 discusses future directions.

A.1 Background: TPTL

The TA-LTL language is based on two existing temporal logics: A-LTL and TPTL [3]. In this section, we briefly overview TPTL.

Temporal logics with a metric for time allow the definition of quantitative temporal relationships (in addition to qualitative ordering), such as distance among events and duration of events. A typical way to add a metric for time in a propositional temporal logic is to replace the unrestricted temporal operators by time-bounded operators. For example, the bounded operator $\Diamond_{[1,3]}\phi$ states "ϕ will become true eventually within 1 to 3 time units from the current time."

An alternative way to represent time quantitatively is to use the *freeze quantification* proposed by Alur and Henzinger [3] in TPTL (Timed Propositional Temporal Logic). TPTL is a formalism for specifying real-time properties. In TPTL, a time variable x can be bound by a freeze quantifier ("$x.$"), which freezes x to the time the temporal context is evaluated. Formally, given a supply of variables $V = \{x, y, z \cdots\}$, a set of atomic propositions $P = \{p, q, \cdots\}$, the syntax of TPTL is inductively defined as follows:

$$\pi := x + c \mid c, \tag{A.1}$$

$$\phi := p \mid \pi_1 \leq \pi_2 \mid \pi_1 \equiv_d \pi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \bigcirc\phi \mid \phi_1 \mathcal{U} \phi_2 \mid x.\phi, \tag{A.2}$$

for $x \in V$, $p \in P$, and $c, d \in \mathbb{N}$, $d \neq 0$.

In TPTL, the timing constraints are of the form $\pi_1 \leq \pi_1$, $\pi_1 \equiv_d \pi_2$. The "$\leq$" operator is the traditional arithmetic inequality operator, and $\pi_1 \equiv_d \pi_2$ means that π_1 is congruent to π_2 modulo the constant d [3]. More notably, the formula $x.\phi$ means that ϕ is *true* if all occurrences of x in ϕ are replaced by the current time τ_0. Other propositional and temporal operators are defined as usual in LTL.

A.2 Timed Adapt Operator-Extended LTL

Next, we describe the syntax and semantics of TA-LTL.

A.2.1 Syntax

TA-LTL is intended to address the *safety*, *liveness*, and *stability* of adaptation behaviors. Safety and liveness are adaptation properties to restrict the duration of adaptation state transitions. Stability constrains the frequency of state changes to ensure a system not to "thrash" by oscillating between states. To support real-time in A-LTL, we adopt the real-time metric in TPTL and apply it to formulae in A-LTL.

The formulae of TA-LTL are constructed from proposition symbols and timing constraints comprising inequality operators, temporal operators, and time reading variables (i.e., variables representing time values). Let P be a set of proposition symbols ($p, q, r, \ldots$), let N be the set of nonnegative integers, and let V be an infinite supply of time reading variables ($x, y, z, \ldots$). The terms of time π and formulae ϕ of TA-LTL are inductively defined as follows:

$$
\begin{aligned}
\pi &:= x + c \mid c \\
\phi &:= p \mid \pi_1 \sim \pi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \phi_1 \Rightarrow \phi_2 \mid \bigcirc\phi \mid \\
&\quad \Box\phi \mid \Diamond\phi \mid \phi_1 \mathcal{U} \phi_2 \mid \phi_1 \overset{\Omega}{\rightarrow} \phi_2 \mid x\ \phi
\end{aligned}
$$

for $x \in V$, $p \in P$, and integer constant $c \in N$.

In TA-LTL, the timing constraints are of the form $\pi_1 \sim \pi_2$, where π_1 and π_2 are two terms, and $\sim$ represents one of the arithmetic inequality operators $<$, $\leq$, $>$, $\geq$, and $=$. The temporal operators $\bigcirc$ (next), $\mathcal{U}$ (until), $\Diamond$ (finally), and $\Box$ (always) are similar to the temporal operators defined in LTL. The operator $\overset{\Omega}{\rightarrow}$ is similar to the

adapt operator defined in A-LTL. "$x.$" is similar to the freeze operator in TPTL, and other propositional operators, such as $\vee$ (or), $\Rightarrow$ (imply), $\Leftrightarrow$ (co-imply), are defined similarly to those in propositional logic.

A.2.2 Semantics of TA-LTL

TA-LTL is evaluated over *timed state sequences*, where each state is an interpretation of a subset of P. A timed state sequence is an infinite sequence of states, each of which is labeled with a discrete time t ($t \in \mathbb{N}$) [3]. Formally, a *state sequence* σ

$$\sigma = \sigma_0\sigma_1\sigma_2\ldots$$

is an infinite sequence of states, where $\sigma_i \subseteq P$ for all $i \geq 0$. A *time sequence*

$$\tau = \tau_0\tau_1\tau_2\ldots$$

is an infinite sequence of time readings, where $\tau_i \in \mathbb{N}$ for all $i \geq 0$, and τ satisfies

Monotonicity: $\tau_i \leq \tau_{i+1}$ for all $i \geq 0$;

Progress: For all $t \in \mathbb{N}$, there is some i ($i \geq 0$) such that $\tau_i > t$.

Thus, a timed state sequence is denoted as a pair $\rho = (\sigma, \tau)$, or it can also be considered as an infinite sequence of pairs $\rho = (\sigma_0, \tau_0), (\sigma_1, \tau_1), \ldots$. We use ρ^i to represent the i^{th} suffix of ρ, i.e., $\rho^i = \rho_i, \rho_{i+1}, \cdots$.

An *environment* ε is an assignment of all the variables in V, which is also extended to be applied to terms. For example, $\varepsilon(x + c)$ is defined to be $\varepsilon(x) + c$. We use $\rho \models_\varepsilon \phi$ to represent the satisfaction relationship between a timed state sequence ρ and a TA-LTL formula ϕ under the environment ε. Similar to A-LTL, we also define TA-LTL semantics in the domain of timed state sequences with finite length. We use $\lambda \models_{fin\varepsilon} \phi$ to represent that a finite timed state sequence λ *satisfies* TA-LTL formula ϕ in the finite sequence domain. The TA-LTL semantics is formally defined as follows:

- $\lambda \models_{fin\varepsilon} \phi$ if and only if $\rho \models_{\varepsilon} \phi$, where ρ is the infinite timed state sequence obtained by extending λ with its last state infinite times. This rule relates the semantics in infinite and finite timed state sequence domains.

- $\rho \models_{\varepsilon} p$ for all $p \in P$, if and only if $p \in \sigma_0$.

- $\rho \models_{\varepsilon} \pi_1 \sim \pi_2$, if and only if $\varepsilon(\pi_1) \sim \varepsilon(\pi_2)$.

- $\rho \models_{\varepsilon} \phi \wedge \psi$, if and only if $\rho \models_{\varepsilon} \phi$ and $\rho \models_{\varepsilon} \psi$.

- $\rho \models_{\varepsilon} \neg\phi$, if and only if $\rho \not\models_{\varepsilon} \phi$.

- $\rho \models_{\varepsilon} \bigcirc\phi$ if and only if $\rho^1 \models_{\varepsilon} \phi$.

- $\rho \models_{\varepsilon} \phi \mathcal{U} \psi$, if and only if there exists i $(i \leq 0)$ such that $\rho^i \models_{\varepsilon} \psi$, and for all j $(0 < j < i)$, $\rho^j \models_{\varepsilon} \phi$.

- $\rho \models_{\varepsilon} x.\phi$, if and only if $\rho \models_{\varepsilon[x:=\tau_0]} \phi$, where $\varepsilon[x := \tau_0]$ is the environment derived by replacing the assignment of x in ε with τ_0.

- $\rho \models_{\varepsilon} \phi \overset{\Omega}{\triangle} \psi$, if and only if there exists i $(i > 0)$ such that $\rho^i \models_{\varepsilon} \psi$ and $\rho_0, \cdots \rho_{i-1} \models_{fin\varepsilon} \phi$.

- Other operators ($\vee$,$\Rightarrow$, $\Diamond$, $\Box$, etc) are defined similarly as those used in LTL.

In this chapter, we assume that all TA-LTL formulae are closed, i.e., any occurrence of time variable x must be within the scope of a freeze operator "$x.$".

A.3 Specifying Adaptation Timing Properties

Figure A.1 depicts three commonly used adaptation scenarios introduced in Chapter 3. Figure A.1(a) depicts the *one-point adaptation* (the source program adapts to the target program at a certain point during its execution); Figure A.1(b)

shows the *guided adaptation* (the source program adapts to the target program with a restriction condition); and Figure A.1(c) depicts the *overlap adaptation* (under the restriction conditions, the target program behavior starts before the source program behavior stops). In this chapter, we focus on specifying the timing properties of these adaptation semantics with TA-LTL by binding time variables to the temporal formulae. Specifically, we investigate three real-time properties of adaptation semantics: *safety*, *liveness*, and *stability*.

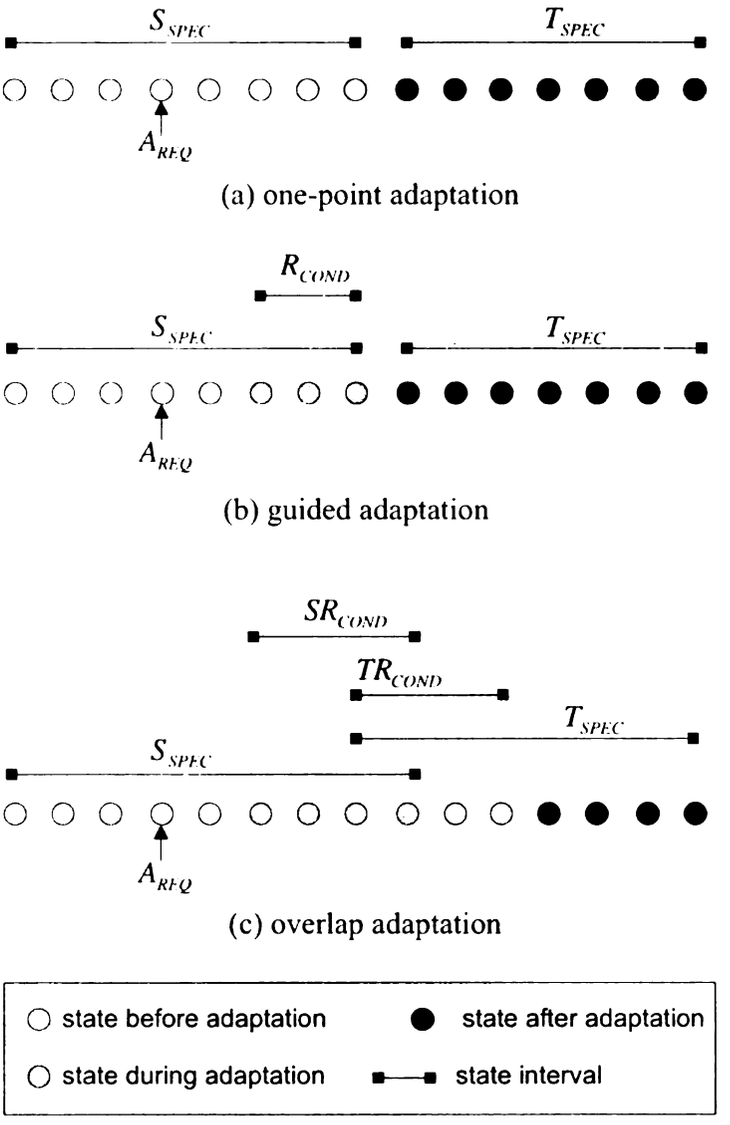


Figure A.1: Three adaptation scenarios

Safety asserts that properties that may invalidate the adaptation will not happen. The timing properties of safety enforce that the source/target program should enter

the restriction condition or safe state within a certain time period after receiving the adaptation request. If this time constraint cannot be satisfied, then it might be too late for the system to react to the dynamically changing environment, attacks, user requirements, etc. Safety properties are most important to adaptations when responding to security threats in which late responses may cause irreversible damage to the system.

Liveness asserts properties that respond to the adaptation requests and asserts that the adaptation goals will eventually be satisfied in time. The liveness timing properties restrict the time periods allowed for the source program to adapt to the target program upon receiving adaptation requests. If this time constraint cannot be satisfied, then the adaptation needs may become invalid or the adaptation may cause the system to enter an inconsistent state [150, 152].

Stability asserts how long the system should remain in certain states. In order to avoid adaptation oscillation caused by frequent adaptation triggering, the timing properties of stability constrain the time interval between two successive adaptations and thus ensure that the system executes in a steady-state program for a certain amount of time before the next adaptation is allowed. If the stability time constraint is violated, then the system might enter an unstable state.

While each steady-state program may have its own real-time constraints, in this chapter, we only focus on adaptation-related real-time constraints and assume that the source and the target steady-state programs have been both specified in LTL. We refer to these specifications as *base specifications* and denote them as S_{SPEC} and T_{SPEC}, respectively. We specify the adaptation from the source program to the target program with A-LTL by extending the base specifications of the source and the target programs. For some adaptations, the source/target program behavior may need to be constrained during the adaptation. We use an LTL formula to specify the *restriction condition*, R_{COND}, during the adaptation.

We use a proposition A_{REQ} to indicate the receipt of an adaptation request to a target program. We identify the fullfilment states for adaptation to be those states where all the obligations required by the source specification S_{SPEC} are fulfilled by the source program, and thus it is safe to terminate the source behavior.

A.3.1 One-Point Adaptation

Under one-point adaptation semantics, after receiving an adaptation request, the source program adapts to the target program at a certain point during its execution. The prerequisite is that the source program should always eventually reach fullfilment states during its execution. This semantics is depicted in Figure A.1(a), where each circle represents a state. Solid lines represent state intervals and the label of each solid line represents the property that is held by the interval. The arrow points to the state in which an adaptation request is received. This adaptation semantics can be specified by A-LTL as follows:

$$\begin{aligned} SPEC_{ONE\text{-}POINT} &= (S_{SPEC} \wedge \Diamond A_{REQ}) \\ &\quad \stackrel{\Omega}{\rightarrow} T_{SPEC}. \end{aligned} \tag{A.3}$$

This formula states that the program initially satisfies S_{SPEC}. When the program reaches a fullfilment state, i.e., all obligations demanded by S_{SPEC} are fulfilled, the program stops being obligated by S_{SPEC} and starts to satisfy T_{SPEC}. In the context of adaptive processing of data streams, an example is to insert the encoder at the sender side where each operation is applied independently to each packet in the outgoing stream.

We specify the liveness timing constraints for one-point adaptation as follows:

$$LIVE_{ONE\text{-}POINT} = \Diamond x.(A_{REQ} \xrightarrow{\Omega} y.T_{SPEC} \wedge (y \leq x + t_{live})). \quad \text{(A.4)}$$

The formula states that once the source program receives an adaptation request (specified by A_{REQ}), it should adapt to the target program (specified by T_{SPEC}) within t_{live}, where t_{live} represents the upper bound of the constrained time frame.

The stability timing constraint for one-point adaptation is specified as follows:

$$STABLE_{ONE\text{-}POINT} = x.((S_{SPEC} \wedge \Diamond A_{REQ}) \xrightarrow{\Omega} y.T_{SPEC} \wedge (y \geq x + t_{freq})). \quad \text{(A.5)}$$

The formula states that the source program should continue to satisfy S_{SPEC} for at least t_{freq} before it can start to satisfy T_{SPEC}.

The timing constraints for one-point adaptation are illustrated in Figure A.2, where the time difference constrained by each formula is denoted by Δt.

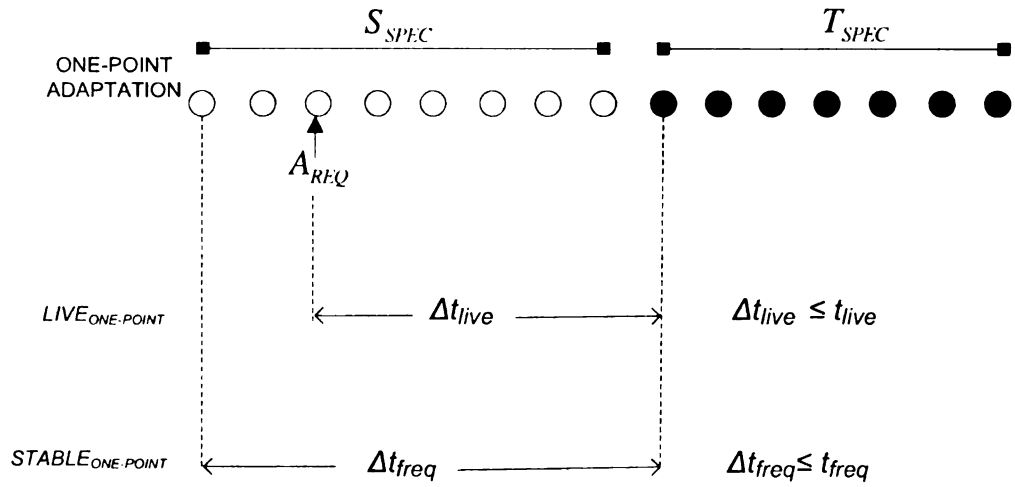


Figure A.2: Real-time constraints for one-point adaptation.

A.3.2 Guided Adaptation

Under guided adaptation semantics (visually depicted in Figure A.1(b)), after receiving an adaptation request, the system first applies a restriction condition to the source program behavior, and then adapts to the target program when it reaches a fullfilment state. This semantics is suited for adaptations where source programs cannot otherwise guarantee reaching a fullfilment state within a given amount of time. The restriction condition ensures that the source program will reach a fullfilment state. This situation arises when the system needs to provide continued service of the old system before switching to the new system. For example, if we are applying encryption encoding/decoding to a sender and a receiver, the receiver needs to process any unencoded packets that are in-flight or buffered at the receiver, before the insertion of the decoder. This adaptation semantics can be specified by A-LTL as follows:

$$\begin{aligned} SPEC_{GUIDED} \;=\; & \big(S_{SPEC} \wedge (\Diamond A_{REQ} \\ & \xrightarrow{\Omega_1} R_{COND})\big) \\ & \xrightarrow{\Omega_2} T_{SPEC}. \end{aligned} \tag{A.6}$$

The specification states that initially S_{SPEC} is satisfied. After an adaptation request, A_{REQ}, is received, the program should satisfy a restriction condition R_{COND} (denoted with $\xrightarrow{\Omega_1}$). When the program reaches a fullfilment state of the source, the program stops being constrained by S_{SPEC}, and starts to satisfy T_{SPEC} (denoted with $\xrightarrow{\Omega_2}$). The *hot-swapping* technique introduced by Appavoo *et al* [5] and the safe adaptation protocol [150, 152] introduced in our previous work use the guided adaptation semantics.

The safety timing property of $SPEC_{GUIDED}$ can be specified by TA-LTL as follows:

$$
\begin{aligned}
SAFE_{GUIDED} \;=\; & \Diamond x.(A_{REQ} \\
& \xrightarrow{\Omega_1} y.R_{COND} \wedge (x + t_{safe} \geq y)) \\
& \xrightarrow{\Omega_2} T_{SPEC}.
\end{aligned} \tag{A.7}
$$

This formula states that upon receiving the adaptation request, the source program should satisfy the restriction condition, R_{COND}, within t_{safe}, otherwise the response is too late to ensure safety in the system, where t_{safe} represents the upper bound of the constrained time frame.

The liveness property $LIVE_{GUIDED}$ for the guided adaptation is specified as follows:

$$
\begin{aligned}
LIVE_{GUIDED} \;=\; & \Diamond x.(A_{REQ} \\
& \xrightarrow{\Omega_1} R_{COND} \\
& \xrightarrow{\Omega_2} y.T_{SPEC} \wedge (y \leq x + t_{live})).
\end{aligned} \tag{A.8}
$$

This formula has a similar meaning to the liveness property for the one-point adaptation. It states that once the source program receives an adaptation request (specified by A_{REQ}), it should adapt to the target program (specified by T_{SPEC}) within t_{live}, where t_{live} represents the upper bound of the constrained time frame.

The stability property $STABLE_{GUIDED}$ for the guided adaptation is as follows:

$$
\begin{aligned}
STABLE_{GUIDED} \;=\; & x.(S_{SPEC} \wedge (\Diamond A_{REQ} \\
& \xrightarrow{\Omega_1} R_{COND})
\end{aligned}
$$

$$\stackrel{\Omega_2}{\rightarrow} y.\, T_{SPEC} \wedge (y \geq x + t_{freq})). \qquad \text{(A.9)}$$

The formula states that the source program should continue to satisfy S_{SPEC} for at least t_{freq} before it can start to satisfy T_{SPEC}.

The real-time constraints for guided adaptation are illustrated in Figure A.3.

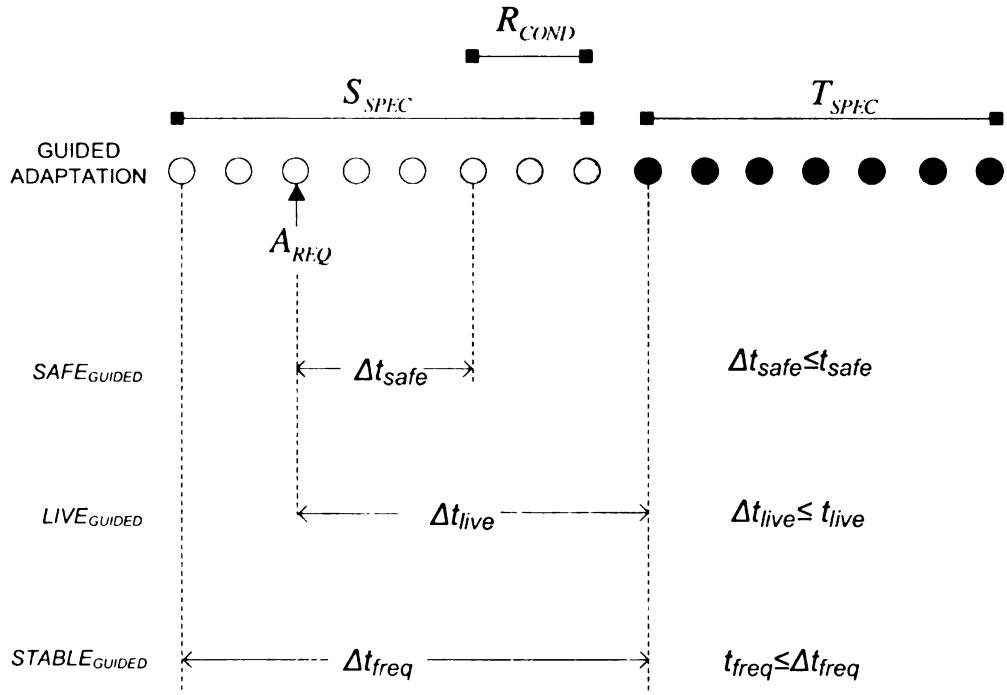


Figure A.3: Real-time constraints for guided adaptation.

A.3.3 Overlap Adaptation

Under overlap adaptation semantics, depicted in Figure A.1(c), the target steady-state program behavior starts before the source steady-state program behavior stops. During the overlap of the source and the target behavior, a restriction condition, R_{COND}, is applied to safeguard the correct behavior of the program. This adaptation semantics is suited for the case when continuous service from the adaptive program is required. The restriction condition ensures that the source program reaches a fullfilment state. For example, in an adaptive communication program, a receiver

may need to be able to handle both unencoded and encoded packets for a period of time after the filter has been reconfigured. The overlap adaptation semantics is formally specified in A-LTL as follows:

$$
\begin{aligned}
SPEC_{OVERLAP} \;=\; & (S_{SPEC} \wedge (\Diamond A_{REQ} \\
& \xrightarrow{\Omega_1} SR_{COND}) \\
& \xrightarrow{\Omega_2} true\,) \\
& \wedge (\Diamond A_{REQ} \\
& \xrightarrow{\Omega_3} T_{SPEC} \wedge (TR_{COND} \\
& \xrightarrow{\Omega_4} true\,)). \qquad \text{(A.10)}
\end{aligned}
$$

This formula states that the entire adaptation process comprises two parts. Initially only the S_{SPEC} is satisfied. After an adaptation request, A_{REQ}, is received, the system should start to satisfy a source restriction condition, SR_{COND} (denoted with $\xrightarrow{\Omega_1}$). When the system reaches a fullfilment state of the source program, the adaptive program stops being obliged by S_{SPEC} and SR_{COND} (denoted with $\xrightarrow{\Omega_2}$). The second part of the formula is used to specify the target program behavior which starts to satisfy T_{SPEC} and target restriction condition TR_{COND} upon receiving adaptation request A_{REQ} (denoted with $\xrightarrow{\Omega_3}$). Finally only the T_{SPEC} continues to be satisfied (denoted with $\xrightarrow{\Omega_4}$). The *graceful adaptation protocol* introduced by Chen *et al* [22] and the distributed reset protocol introduced by Kulkarni *et al* [77] use the overlap adaptation semantics.

The real-time properties of $SPEC_{OVERLAP}$ are shown in Formulae A.11–A.16 as follows:

$$SAFE_{REQ\text{-}ADAPT} = \Diamond x.(A_{REQ} \xrightarrow{\Omega_1} y.SR_{COND} \wedge (x + t_{s-ra} \geq y)) \xrightarrow{\Omega_2} true, \quad \text{(A.11)}$$

$$LIVE_{REQ\text{-}FULFILL} = \Diamond x.(A_{REQ} \xrightarrow{\Omega_1} SR_{COND} \xrightarrow{\Omega_2} y.(x + t_{l-rf} \geq y)), \quad \text{(A.12)}$$

$$LIVE_{REQ\text{-}TSTART} = \Diamond x.(A_{REQ} \xrightarrow{\Omega_3} y.(T_{SPEC} \wedge TR_{COND} \xrightarrow{\Omega_4} (x + t_{l-rt} \geq y))), \quad \text{(A.13)}$$

$$LIVE_{TSTART\text{-}COMPLETE} = \Diamond A_{REQ} \xrightarrow{\Omega_3} (x.T_{SPEC} \wedge (TR_{COND} \xrightarrow{\Omega_4} y.(x + t_{l-tc} \geq y))), \quad \text{(A.14)}$$

$$LIVE_{REQ\text{-}COMPLETE} = \Diamond x.(A_{REQ} \xrightarrow{\Omega_3} T_{SPEC} \wedge (TR_{COND}$$

$$\stackrel{\Omega_1}{\rightarrow} y.(x + t_{l-rc} \geq y))). \quad \text{(A.15)}$$

$$\begin{aligned} STABLE_{OVERLAP} \quad = \quad & x.(S_{SPEC} \wedge \Diamond A_{REQ} \\ & \stackrel{\Omega_1}{\rightarrow} SR_{COND} \\ & \stackrel{\Omega_2}{\rightarrow} y.(x + t_{freq} \leq y)). \end{aligned} \quad \text{(A.16)}$$

In Formula A.11, $SAFE_{REQ\text{-}ADAPT}$ states that upon receiving the adaptation request, the source program should satisfy the source restriction condition SR_{COND} within t_{s-ra}, otherwise the response time is too long to be effective. In Formula A.12, $LIVE_{REQ\text{-}FULFILL}$ states that once the source program receives an adaptation request, it should complete the whole adaptation process within t_{l-rf}. Similarly, in Formula A.13, $LIVE_{REQ\text{-}TSTART}$ indicates that once the adaptation request is received, the target program must start within t_{l-rt}. In Formula A.14, $LIVE_{TSTART\text{-}COMPLETE}$ states that once the target behavior starts, the entire adaptation process must complete with t_{l-tc}. In Formula A.15, $LIVE_{REQ\text{-}COMPLETE}$ states the time from receiving the adaptation request to being fully functioning must be within t_{l-rc}. And finally, in Formula A.16, $STABLE_{OVERLAP}$ states that the source program should continue to satisfy S_{SPEC} for at least t_{freq} before it may stop satisfying S_{SPEC}. The real-time constraints for overlap adaptation are illustrated in Figure A.4.

A.4 Case Study: Live Audio Streaming Program

To illustrate the use of TA-LTL in specifying adaptation timing properties, we use TA-LTL to specify the safety, liveness, and stability of the adaptation behavior of an adaptive audio streaming system. The interconnection of the systems is depicted in Figure A.5. A live audio stream is multicast from a wired desktop computer to

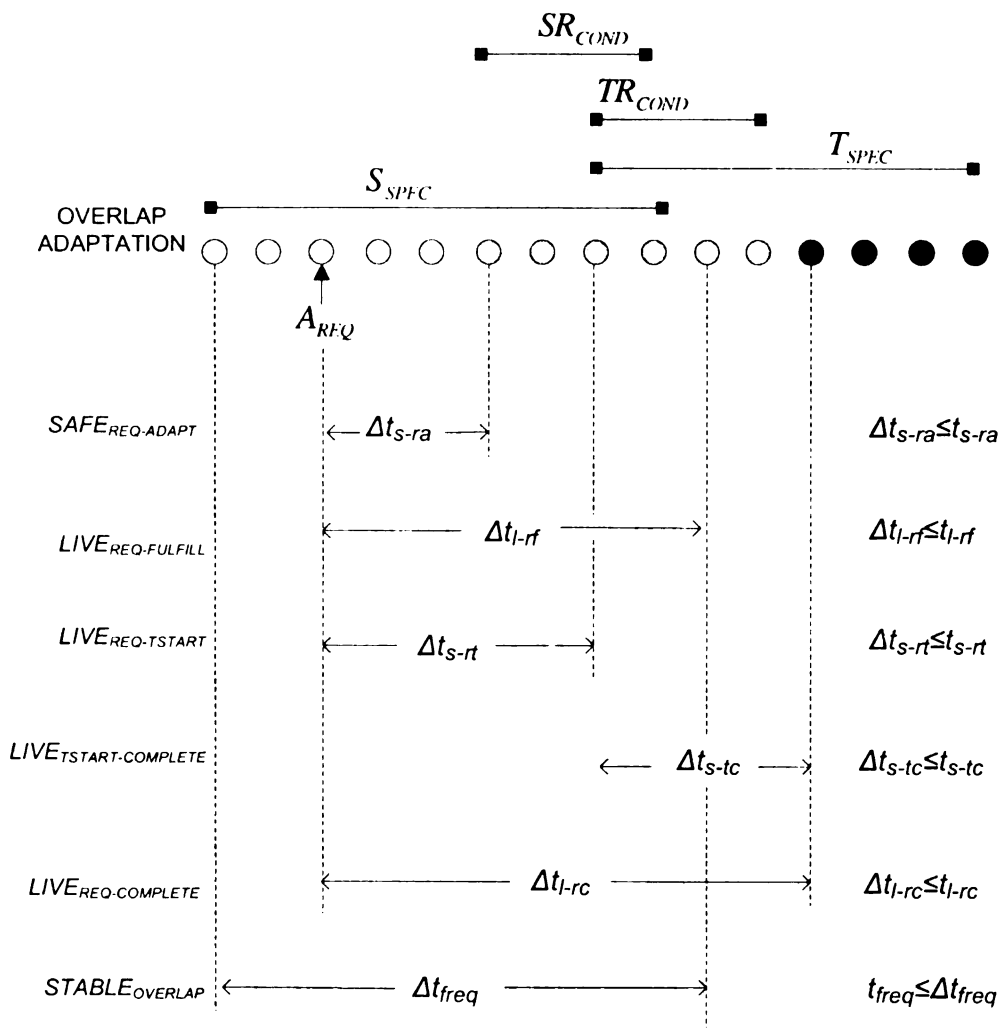


Figure A.4: Real-time constraints for overlap adaptation.

multiple mobile devices via the 802.11b WLAN. Effectively, the receivers are used as multicast-capable Internet "phones" participating in a conferencing application. We expect the system to operate in environments with different packet loss rates. 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 or should remove the encoder/decoder completely; when the loss rate is high, they should use a high-loss-tolerance and consequently high bandwidth-consuming encoder/decoder.

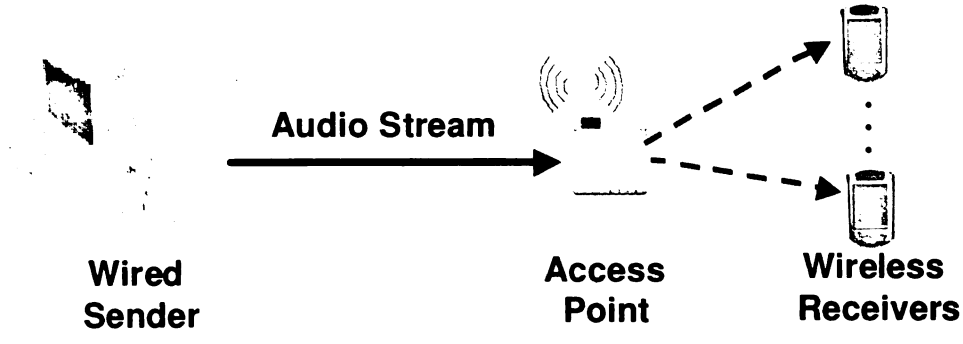


Figure A.5: Audio streaming system connection

A.4.1 Forward Error Correction (FEC) Filters

The adaptive behavior in this system is realized using MetaSockets [120] (Section 3.5). In this case study, we use two different FEC filters to respond to the dynamically changing loss rate. One filter uses block erasure codes [116], and the other uses the GSM 06.10 encoding algorithm also used for cellular telephones [34]. The FEC encoded data are different from the original audio data and can be played only after they are decoded by the FEC decoder. A run-time scenario is to have all collected audio samples at the sender side stored in a data buffer waiting to be encoded by the FEC encoder. The encoded data packets are then multicast to the receivers and stored in a data buffer waiting to be decoded by the FEC decoder. The decoded audio data will eventually be played by the receivers. Here, we use the

guided adaptation semantics to specify the adaptation of inserting an FEC filter at the receiver side.

The use of (n, k) *block erasure codes* for error correction was popularized by Rizzo [116] and is now used in many wired and wireless distributed systems. Figure A.6 depicts the basic operation of these codes. An encoder converts k source packets into n encoded packets, such that any k of the n encoded packets can be used to reconstruct the k source packets [116]. In this chapter, we use only *systematic* codes, which means that the first k of the n encoded packets are identical to the k source packets. We refer to the first k packets as *data* packets, and the remaining $n - k$ packets as *parity* packets. Each set of n encoded packets is referred to as a *group*. The advantage of using block erasure codes for multicasting is that any x $(x \leq n - k)$ parity packet can be used to correct independent x packet losses among different receivers [116]. In the remainder of the chapter, we refer to the block-oriented FEC simply as "FEC (n, k)". While block-oriented FEC approaches are effective in improving the quality of interactive audio streams on wireless networks, the group sizes must be relatively small in order to reduce playback delays.

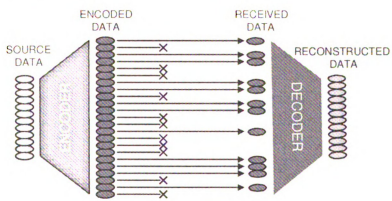
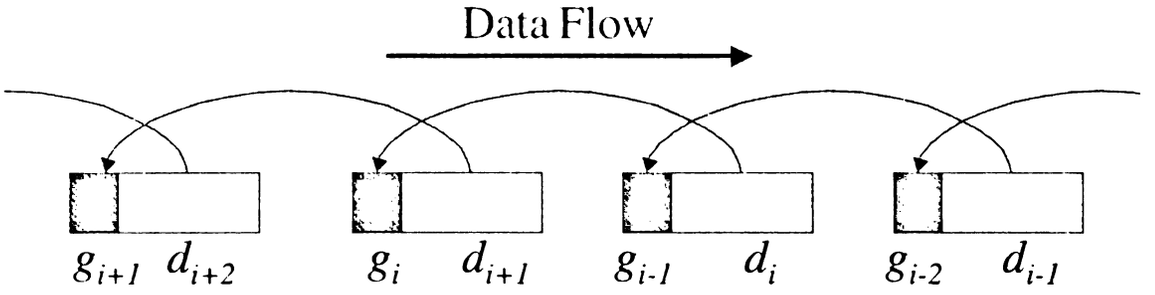


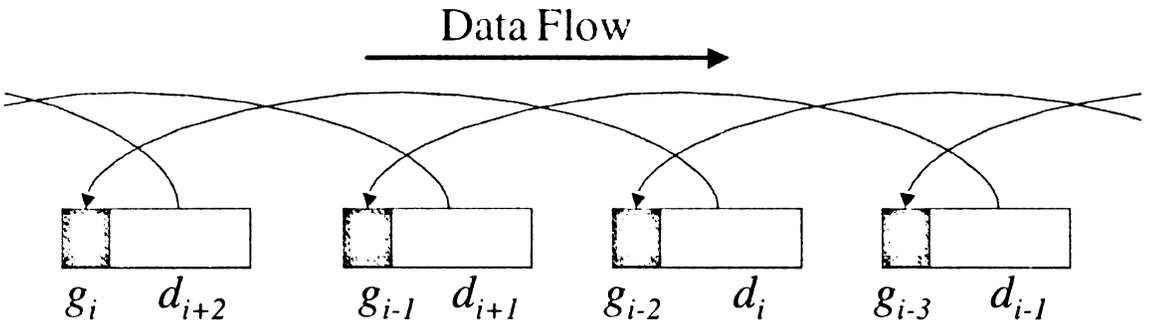
Figure A.6: Operation of block erasure code.

An alternative approach with lower delay and lower overhead is the *GSM-oriented FEC encoding* (also known as *signal processing-based FEC* (SFEC)) [16], in which a lossy, compressed encoding of each packet p_i is piggybacked onto one or more

subsequent packets. If packet p_i is lost, but one of the encodings of packet p_i arrives at the receiver, then at least a lower quality version of the packet can be played to the listener. The parameter θ is the offset between the original packet and its compressed version. Figure A.7 shows two different examples, one with $\theta = 1$ and the other with $\theta = 2$. It is also possible to place multiple encodings of the same packet in multiple subsequent packets, such as using both $\theta_1 = 1$ and $\theta_2 = 3$. Although GSM is a CPU-intensive coding algorithm [16], the bandwidth overhead is very small. In the remainder of the chapter, we refer to the GSM-oriented FEC simply as "GSM (θ, c)," which means copies of the coded packet p are placed in c successive packets, beginning from the θ^{th} packet after p.



(a) GSM encoding with $\theta = 1$



(b) GSM encoding with $\theta = 2$

Figure A.7: GSM encoding on a packet stream (d_i: data, g_i: copy).

A.4.2 Specifying QoS Constraint with TA-LTL

In previous research [154], Zhou *et al* have investigated the quality-of-service (QoS) properties (such as packet delivery rate and delay) of these two FEC protocols. Table A.1 shows the loss rate perceived by the receiving application, that is, *after FEC decoding*, for different FEC (n, k) or GSM (θ, c) settings. Another factor important to real-time communication is the additional delay introduced into the packet stream by the encoding and decoding filters. Table A.2 shows the worst case delay introduced by different FEC codes to wait for the encoded packets. For example, considering FEC (8, 4) and GSM (3, 1), if the first data packet is lost, then the receiver will need to wait for at least 3 packets until the first parity packet or piggybacked packet arrives to recover the loss.

Table A.1: Loss rate comparison of different FEC codes

Code	Raw	GSM (1,1)	GSM (1,2)	GSM (1,3)	GSM (2,1)	GSM (2,2)	GSM (2,3)
%	28	11.05	6.28	3.53	13.88	6.8	3.78
Code	------	GSM (3,1)	GSM (3,2)	GSM (3,3)	GSM (4,4)	GSM (6,4)	GSM (8,4)
%	------	10.27	4.8	2.75	28.20	16.29	9.16

Table A.2: Delay comparison of different FEC codes

Code	Raw	GSM (1,1)	GSM (1,2)	GSM (1,3)	GSM (2,1)	GSM (2,2)	GSM (2,3)
%	0	19.95	39.95	59.9	40.83	60.37	80.10
Code	------	GSM (3,1)	GSM (3,2)	GSM (3,3)	GSM (4,4)	GSM (6.4)	GSM (8,4)
%	------	62.55	82.52	102.87	17.79	37.73	52.33

From the QoS comparison we know that for a certain wireless environment (e.g.,

28% raw loss rate as shown in Table A.1), different FEC codes have different loss recovery performance while holding different timing characteristics. For example, if the raw loss rate of the wireless environment changes to 28% and the expected loss rate after FEC decoding is $\leq 10\%$, FEC (8, 4), GSM (1, 2), GSM (1, 3), GSM (2, 3), GSM (2, 3), GSM (3, 2), and GSM (3, 3) are possible filter candidates for adaptation. However, in order to satisfy the real-time interactive audio streaming requirements, the accumulated delay after adaptation should not exceed 150 milliseconds [118]. For this example, in the guided adaptation, the restriction condition is that the data is blocked from transmission during the adaptation. Thus the accumulated delay comprises two parts: the adaptation delay and the FEC delay. If there is no additional operation (e.g., dropping packet, fast playback) to reduce the delay, this accumulated delay will remain in the post-adaptation communication. Thus we can use TA-LTL to specify the real-time constraint, and the selection of FEC filters and the implementation of adaptation should satisfy this constraint.

This adaptation semantics and its timing properties can be specified by A-LTL and TA-LTL as shown in Figure A.8. In Formula A.17, $SPEC_{FEC}$ states that initially the system is running without FEC filters (the source program), i.e., it satisfies $NoFEC_{SPEC}$. After an adaptation request, A_{REQ}, is received, the system should satisfy a restriction condition R_{COND}, which requires the receiver to temporarily stop receiving incoming encoded FEC data. When the source program reaches a fullfilment state, i.e., all original audio data in the buffer have been played, the system stops satisfying the $NoFEC_{SPEC}$, and starts to satisfy $WithFEC_{SPEC}$. In Formula A.18, $SAFE_{FEC}$ states that upon receiving the FEC decoder insertion adaptation request, the source program should stop receiving incoming packets within t_{safe}, otherwise the encoded FEC data will be received and stored in the data buffer, causing a failure since it cannot be played directly. In Formula A.19, $LIVE_{FEC}$ specifies the real-time constraint of the adaptation for inserting FEC filters discussed above. Specifically,

t_{live} is the criteria for ensuring real-time audio communication (150 milliseconds in this study). In Formula A.20, $STABLE_{FEC}$ prevents the system from inserting/removing the FEC filter too frequently. This example illustrates the use of TA-LTL in specifying quantitative timing requirements for adaptive communication software.

$$\begin{aligned} SPEC_{FEC} &= NoFEC_{SPEC} \wedge (\Diamond A_{REQ} \\ &\quad \xrightarrow{\Omega_1} R_{COND}) \\ &\quad \xrightarrow{\Omega_2} WithFEC_{SPEC}, \end{aligned} \tag{A.17}$$

$$\begin{aligned} SAFE_{FEC} &= \Diamond x.(A_{REQ} \\ &\quad \xrightarrow{\Omega_1} y.R_{COND} \wedge (x + t_{safe} \geq y)) \\ &\quad \xrightarrow{\Omega_2} WithFEC_{SPEC}, \end{aligned} \tag{A.18}$$

$$\begin{aligned} LIVE_{FEC} &= \Diamond x.(A_{REQ} \\ &\quad \xrightarrow{\Omega_1} R_{COND} \\ &\quad \xrightarrow{\Omega_2} y.\, WithFEC_{SPEC} \wedge (y \leq x + t_{live})), \end{aligned} \tag{A.19}$$

$$\begin{aligned} STABLE_{FEC} &= x.(NoFEC_{SPEC} \wedge (\Diamond A_{REQ} \\ &\quad \xrightarrow{\Omega_1} R_{COND}) \\ &\quad \xrightarrow{\Omega_2} y.\, WithFEC_{SPEC} \wedge (y \geq x + t_{freq})). \end{aligned} \tag{A.20}$$

Figure A.8: Specifying adaptation semantics with TA-LTL

A.5 Related Work

Temporal logics have been extended in many different ways to support the specification of quantitative timing properties [3, 31, 48, 107, 113], including the TPTL [3] introduced in Section A.1 and FLEA [31] introduced in Section 3.6. Most of the dis-

cussion in the related work section for A-LTL (Section 3.6) applies to TA-LTL as well. Similar to the discussion for A-LTL, none of the above real-time logics is designed specifically for specifying real-time adaptation properties as TA-LTL described in this Chapter. TA-LTL differs from all other real-time temporal logics by its convenience of specifying real-time constraints in adaptation semantics.

A.6 Discussion

In this section, we discuss the expressiveness, decidability, and model checking algorithms of TA-LTL.

A.6.1 Expressiveness

The TA-LTL introduced in this chapter can be considered a combination of A-LTL and TPTL. A-LTL is a non-real time temporal logic, and thus it does not reference τ, the time sequence, in a timed state sequence. Therefore, TA-LTL is strictly more expressive than A-LTL.

Theorem 14: *TA-LTL is strictly more expressive than A-LTL.*

Proof 14:

We prove that TA-LTL is more expressive than A-LTL but not vice-versa.

1. *TA-LTL is more expressive than A-LTL: TA-LTL* $\geq$ *A-LTL.*

   The proof of this statement is straightforward. Since TA-LTL is a super set of A-LTL, for any formula ϕ *of A-LTL, there is also a formula* ϕ' *of exactly the same form in TA-LTL, and* $\phi' = \phi$*. Therefore, we have TA-LTL* $\geq$ *A-LTL.*

2. *A-LTL is not more expressive than TA-LTL: A-LTL* $\ngeq$ *TA-LTL.*

For the timed state sequences

$\sigma = (\neg p, 0), (p, 1), (p, 2), \cdots$ *and*

$\sigma' = (\neg p, 0), (p, 5), (p, 6), \cdots$.

there is a formula in TA-LTL, $\phi = \Diamond x.p \wedge (x < 5)$, that distinguishes these two sequences. That is, $\sigma \models_{inf} \phi$, while $\sigma' \not\models_{inf} \phi$. However, for any formula ψ in A-LTL, we have $\sigma \models \psi$ if and only if $\sigma' \models \psi$. Therefore, there is no formula in A-LTL that accepts exactly the same set of sequences as ϕ does. Thus we have A-LTL $\not\geq$ TA-LTL.

Therefore, TA-LTL > A-LTL.

A.6.2 Decision Procedure and Model Checking

For a given logic, a formula ϕ of the logic is *satisfiable* if and only if there exists a model σ, such that $\sigma \models \phi$. A *decision procedure* of a logic determines whether each formula of the given logic is satisfiable. Tableaux-based decision procedures have been adopted by researchers to address the satisfiability problem of temporal logics including LTL [88], the choppy logic [119], etc. In Chapter 7 we also proposed using a tableaux-based decision procedure for the satisfiability problem for A-LTL [146]. Alur *et al.* [3] extended the traditional tableaux-based approach with time-difference propositions to address the real-time properties in TPTL satisfiability problem. The same technique can be used to extend the A-LTL decision procedure for TA-LTL.

Model checking techniques determine whether a program satisfies a given temporal logic formula by exploring the state space of the program. We adopt the timed state graph [3] as the formal model for real-time programs. A timed state graph is a tuple $T = \langle L, \mu, \nu, L_0, \mathfrak{T} \rangle$, where

- L is a finite set of locations;

- $\mu : L \rightarrow 2^P$ is a labeling function that labels each location $l \in L$ with a state $\mu_l \subseteq P$;
- ν is a time difference function that labels each location with the time different from its predecessor location;
- $L_0 \subseteq L$ is a set of initial locations;
- $\mathfrak{T} \subseteq L^2$ is a set of transitions.

A computation of the timed state graph is a timed state sequence $\rho = (\sigma, \tau)$ if and only if there is an infinite path $l = l_0, l_1, \cdots$ of T, such that $\sigma_i = \mu_{l_i}$, and the time difference in $l_i(+1)$ equals $(\tau_{i+1} - \tau_i)$. The model checking of a timed state graph T against a TA-LTL formula ϕ proceeds as follows:

1. Calculate the negation $\neg\phi$ of the formula ϕ;
2. construct the tableau $T(\neg\phi)$ for $\neg\phi$;
3. calculate the product $P = T * T(\neg\phi)$;
4. determine whether the product P contains an accepting path σ starting from a state that contains a initial state of T and $\neg\phi$.

The program model T satisfies the formula ϕ if and only if there does not exist such a path σ in P.

A.6.3 Summary

This chapter introduced the specification of adaptation timing properties in autonomic systems in terms of TA-LTL. After an adaptation temporal specification is constructed, it may serve as guidance for the adaptation developers to clarify the intent for the adaptive program. It can also be used to check for consistency in the

temporal logic specifications. By using run-time verification techniques, we may also automatically identify the safe states for an adaptation and insert adaptation logic at appropriate points in the program. We may even perform model checking to verify the correctness of the program model against the temporal logic specifications.

Appendix B

Supporting Material for MASD

B.1 Rapid Prototyping for Adaptive GSM-Oriented Protocol

We include the Java code for the rapid prototyping example we introduced in Section 5.3.1. The prototype uses sockets to transmit packets from the sender to the receiver. `Prototype.java` instantiates the sender, receiver, and internet, and defines the relationships among them. `Sender.java` defines the functions to be invoked by the GSM sender adaptation net in Chapter 5 (Figure 5.10). `Receiver.java` defines the functions to be invoked by the GSM receiver adaptation net in Chapter 5 (Figure 5.11). `Internet.java` defines the functions to be invoked by the GSM lossy network net (environment model) in Chapter 5 (Figure 5.7).

Prototype.java

```
/*****************************************
 *  instantiates the sender, receiver, and  *
 *  internet, and defines the relationships *
 *  among them.                             *
 *****************************************/

package gsm;

public class Prototype {

  static public Sender sender;
  static public Receiver receiver;

  static public Internet inet;

  static public void main(String[] args){
    inet  = new Internet();
    receiver = new Receiver(inet);
    inet.initReceiver();
    receiver.receive();
  }

  //initialize sender and receiver
  static public void init(){
    inet = new Internet();
    sender = new Sender(inet);
    receiver = new Receiver(inet);
  }

  //initialize sender object
  static public void initSender(){
    inet = new Internet();
    sender = new Sender(inet);
    inet.initSender();
  }

  //initialize receiver object
  static public void initReceiver(){
    inet  = new Internet();
    receiver = new Receiver(inet);
    inet.initReceiver();
  }

  initialize the internet object.
  static public void initInet(){
```

```
    inet = new Internet();
  }
}
```

Sender.java

```
/*****************************************
 * The functions implemented in this class   *
 * are to be invoked by the GSM sender       *
 * adaptation net in Chapter 5 (Figure 5.10).*
 *****************************************/

package gsm;

import java.util.*;

public class Sender{

  final int SIZE = 10;
  int index =1;
  byte[] dataInput;
  LinkedList gsmData = new LinkedList();
  ArrayList gsmPacket = new ArrayList();
  Internet inet;
  Random ran = new Random();
  int simulateInput = 1;
  int para;
  SenderNet net = new SenderNet();


  static public void main(String[] args){
    Prototype.initSender();
    Prototype.sender.start();
  }

  static public void test(){
    SenderNet net = new SenderNet();
    net.init();
    net.readdata();
    net.S_encode();
    net.S_send();
  }


  public void start(){
    para = 2;
    while(true){
      readData();
      packetGen(1,para);
      if(ran.nextFloat() > 0.95)  break;
      sendData();
```

```
    }

    adapt();
    para = 3;
    sendData();

    while(true){
      readData();
      packetGen(1,para);
      sendData();
    }
  }

  public Sender(Internet internet){
    net.init();
    inet = internet;
    gsmData.addLast(null);
    gsmData.addLast(null);
  }


  public void readData(){
    net.readdata();
    simulateInput ++;
    gsmData.addLast(GSM.encode(dataInput));
    try{
      Thread.sleep(300);
    }catch(Exception e){
      System.out.println("exception during sleep)");
    }
  }


  public void packetGen(int i,int j){
    if(para == 2){
      net.S_encode();
    }
    else {
      net.T_encode();
    }
    gsmPacket = new ArrayList();
    gsmPacket.add(new Integer(index));
    gsmPacket.add(dataInput);

    //if j =2 then we should add 1, 0
    for (int k = 0;k<j;k++)
      gsmPacket.add (gsmData.get(j-1-k));
```

```
    gsmData.remove(0);
    index ++;

  }

  public void sendData(){
    if(ran.nextFloat() > 0.6) {
      if(para == 2) {
        net.S_lose();
      }
      else{
        net.T_lose();
      }
      System.out.println("losing  "+ (index-1));
    }
    else{
      if(para == 2) {
        net.S_send();
      }
      else {
        net.T_send();
      }
      inet.send(gsmPacket);
    }
  }

  public void adapt(){
    gsmData.addFirst(gsmPacket.get(1));
    gsmPacket.add(null);
    net.adapt();
  }
}
```

Receiver.java

```
/*****************************************
* The functions implemented in this class   *
* are to be invoked by the GSM receiver     *
* adaptation net in Chapter 5 (Figure 5.11).*
*****************************************/

package gsm;

import java.util.*;

public class Receiver{
  ArrayList receiveDataBuffer = new ArrayList(16);
  ArrayList gsmPacket=null;
  int[] index = new int[16];
  Internet inet;

  static public void main(String[] args){
    int n;
    Sample.initReceiver();
    n=2;
    Sample.receiver.receive();
    while (true) {
      while(true) {
        Sample.receiver.decode(0);
        Sample.receiver.decode(1);
        Sample.receiver.putData();
        Sample.receiver.decodeLast(2);
        if (Sample.receiver.getPacketIndex() == 0)
          break;
      }
      Sample.receiver.receive();
      if(Sample.receiver.packetLength() !=4)
        break ;
    }
    if (Sample.receiver.packetLength() == 5)
      Sample.receiver.adapt(3);
    n = 3;
    while (true)  {
      while(true)  {
        Sample.receiver.decode(0);
        Sample.receiver.decode(1);
        Sample.receiver.decode(2);
        Sample.receiver.putData();
        Sample.receiver.decodeLast(3);
        if (Sample.receiver.getPacketIndex() == 0)
```

```
        break;
      }
      Sample.receiver.receive();
      if(Sample.receiver.packetLength() !=5)
        break;
    }
  }


  public Receiver(Internet internet){
    for(int i = 0;i<16;i++){
        index[i]=i-2;
        receiveDataBuffer.add(null);
    }
    inet = internet;
  }

  public void receive(){
    gsmPacket = inet.receive();
  }

  public int getPacketIndex(){
    if (gsmPacket  == null) return 0;
    else
      return ((Integer) gsmPacket.get(0)).intValue();
  }

  public void putData(){
    byte[] data  = (byte[]) receiveDataBuffer.get(0);
    if(data ==null)
      System.out.println("losing "+index[0]+" "+gsmPacket.size());
    else
      System.out.println(new String(data)+" "+gsmPacket.size());
  }

  public void  decode(int j){
    int k,i;
    byte[] s,r;
    k = index[j+1];
    s=  (byte[]) receiveDataBuffer.get(j+1);
    if (gsmPacket == null) {
      r=s;
      i = 0;
    }
    else{
      i = ((Integer) gsmPacket.get(0)).intValue();
      if(k<i-2 || k > i)    r=s;
```

```
      else r = (byte []) GSM.decode((byte[])gsmPacket.get(i-k+1));
    }
    receiveDataBuffer.set(j,r);
    index[j] = k;
  }

  public void decodeLast(int j){
    int k,i;
    byte[] s,r;
    k = index[j]+1;
    s=  (byte[]) receiveDataBuffer.get(j+1);
    if (gsmPacket == null) {
      r=s;
      i = 0;
    }
    else{
      i = ((Integer) gsmPacket.get(0)).intValue();
      if(i>k+2 || k > i) r=s;
      else
        r = (byte []) GSM.decode((byte[])gsmPacket.get(i-k+1));
    }
    receiveDataBuffer.set(j,r);
    index[j] = k;
    if(i<=k)
      gsmPacket = null;

  }

  public int packetLength(){
    if (gsmPacket == null)
      return 0;
    else
      return gsmPacket.size();
  }

  public void adapt(int par){
    if (par == 3){
      for(int i = par;i>0;i--)
        index[i]=index[i-1];
      index[0]=index[1]-1;
      receiveDataBuffer.add(0,null);
    }
  }
}
```

Internet.java

```
/*****************************************
* The functions implemented in this class   *
* are to be invoked by the Lossy network net*
* in Chapter 5 (Figure 5.7).                *
*****************************************/

package gsm;

import java.util.*; import java.net.*; import java.io.*;

public class Internet {
  LinkedList buffer = new LinkedList();

  public Socket senderSocket;
  public Socket receiverSocket;
  public ServerSocket receiverServerSocket;
  public ObjectOutputStream  senderOut = null,
  public ObjectInputStream senderIn = null;
  public ObjectOutputStream  receiverOut = null;
  public ObjectInputStream receiverIn = null;

  public synchronized void send(ArrayList gsmPacket){
      try{
        senderOut.writeObject(gsmPacket);
      }catch(Exception e){
        System.out.println("exception occurs when write");
      }
  }

  public synchronized  ArrayList receive(){
      ArrayList ret = null;
      try{
        ret = (ArrayList) receiverIn.readObject();
      }
      catch(Exception e){
    System.out.println("exception occurs when receive");
      }
      return  ret;
  }

  public synchronized void dataLoss(){
  }

  public void initSender(){
```

```
    try{
      senderSocket = new Socket("localhost", 1234);
      senderOut = new ObjectOutputStream(senderSocket.getOutputStream());
      senderIn = new ObjectInputStream(senderSocket.getInputStream());
    }catch(Exception e){
      System.out.println("exception happens when initiating sender");
    }
  }

  public void initReceiver(){
    try{
      receiverServerSocket = new ServerSocket(1234);
    }catch(Exception e){
      System.out.println("exception creating receiver server socket");
    }
    try{
      receiverSocket= receiverServerSocket.accept();
    }catch(Exception e){
      System.out.println("exception when accepting a connection");
    }
    try{
      receiverOut =
        new ObjectOutputStream(receiverSocket.getOutputStream());
      receiverIn =
        new ObjectInputStream(receiverSocket.getInputStream());
    }catch(Exception e){
      System.out.println("exception creating output input streams");
    }
  }
}
```

B.2 Stub Files for Model-based Testing

In this section, we include the Renew stub files used to generate handler methods for the transitions in the Petri net adaptation models introduced in Section 5.3.2. The file `ReceiverNet.stub` defines handler functions for the transitions in the receiver adaptation net in Figure 5.11. The file `SenderNet.stub` defines handler functions for the transitions in the sender adaptation net in Figure 5.10.

ReceiverNet.stub

```
/*********************************************************
*  The ReceiverNet stub file defines handler functions   *
*  for the transitions  in the receiver adaptation net   *
*  in Figure 5.11. The handler functions will be         *
*  generated in ReceiverNet.java.                        *
*********************************************************/


package gsm;

class ReceiverNet for net gsmreceiver {
  void init() {
    this:init();
  }


  void writedata(){
    this:writedata();
  }

  void S_decode(){
    this:S_decode();
  }


  void S_receive(){
    this:S_receive();
  }

  void adapt(){
    this:adapt();
  }

  void T_decode(){
    this:T_decode();
  }

  void T_receive(){
    this:T_receive();
  }
}
```

SenderNet.stub

```
/*********************************************************
*  The SenderNet stub file defines handler functions     *
*  for the transitions  in the sender adaptation net     *
*  in Figure 5.10. The handler functions will be         *
*  generated in SenverNet.java.                          *
*********************************************************/


package gsm; class SenderNet for net gsmsender {
  void init() {
    this:init();
  }

  void readdata(){
    this:readdata();
  }


  void S_encode(){
    this:S_encode();
  }


  void S_lose(){
    this:S_lose();
  }


  void S_send(){
    this:S_send();
  }


  void adapt(){
    this:adapt();
  }


  void T_lose(){
    this:T_lose();
  }
```

```
  void T_encode(){
    this:T_encode();
  }


  void T_send(){
    this:T_send();
  }
}
```

Appendix C

Obtaining Statechart Diagrams from Java Code

In this chapter, we include the metamodels and the translation rules for the Java to UML translation used in Chapter 6. Figure C.1 and Figure C.2, respectively, show the metamodels for the subset of Java legacy programs and the subset of UML Statechart diagrams that we currently support. The metamodel-based translation rules are summarized in Table C.1. We now describe the translation of each feature in Java programs.

- Java legacy program: We translate a Java legacy program into a UML Statechart model.
- Java class: Each java class is represented as a state machine describing its behavior. Note that for each instance of the class, we need to create a separate instance of the state machine.
- Constructor method: The constructor method of a class is modeled as a transition from the initial state of the state machine to a state labeled new object. The new object state represents the state in which no method of the class is

Java Program Features	UML Statechart Diagram Features
Java legacy program	UML Statechart model
Java class	State machine
Constructor method	Transition from the initial state to a state labeled "new object"
Method	Submachine
Atrributes	Attribute
Condition	Condition
Simple block	Transition with local action
If-else block	A combination of states and branching transitions
Loop block	A combination of states and transitions with a transition going back to the entry state
Sequential block	A sequence of states and transitions
Statement	Local action
Entry point of simple block	Incoming state of the transition for the block
Exit point of simple block	Outgoing state of the transition for the block
Entry point of method call with object `Obj` and method `M`	A state with outgoing transition T, where T includes a send event action with target `Obj` and event `M`
Exit point of method call with method M	A State with outgoing transition triggering event M_RETURN
Entry point of a synchronized block	A substatemachine that waits for a monitor and sets the monitor
Exit point of a synchronized block	A state with an outoing transition action releasing the monitor

Table C.1: Rules for the Metamodel-based Java to UML translation

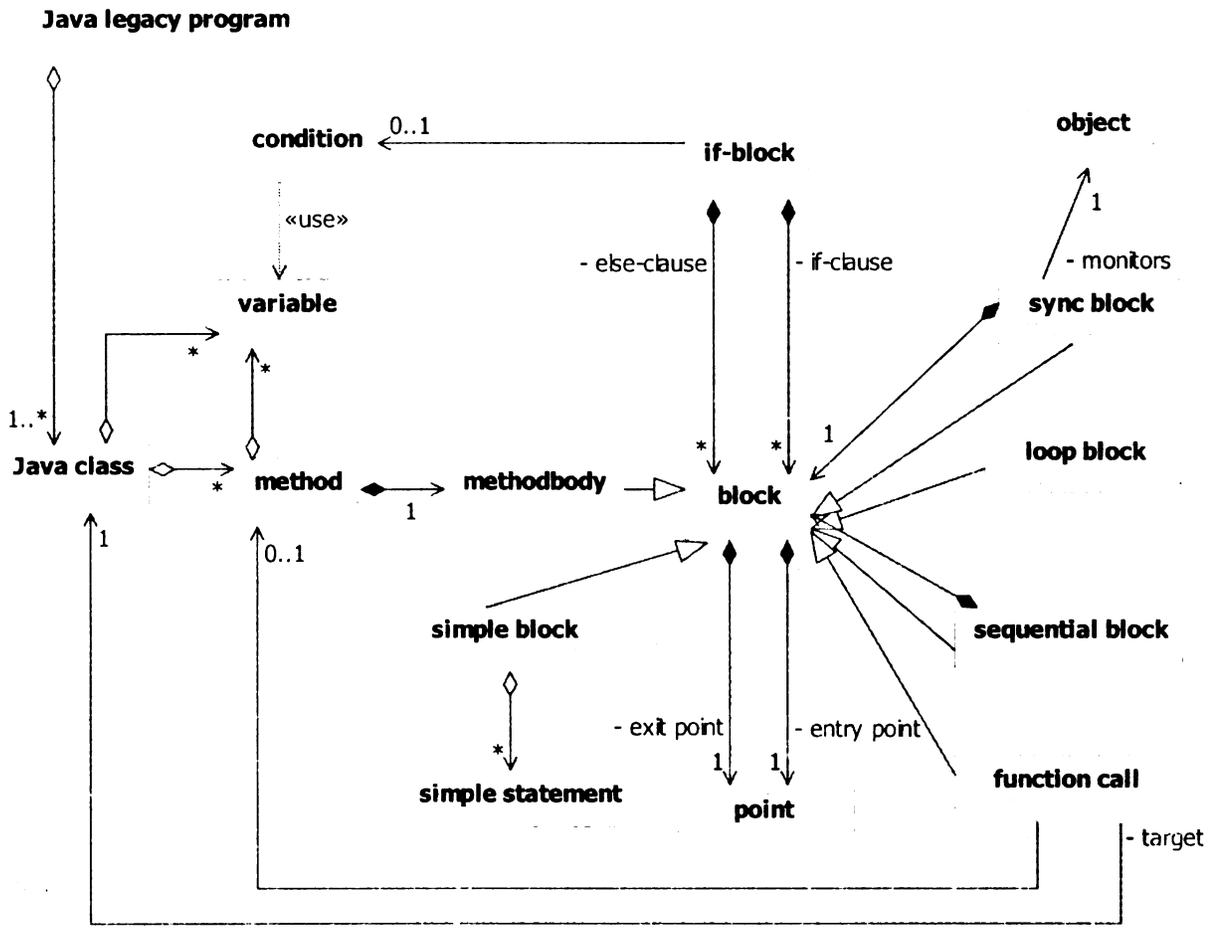


Figure C.1: Metamodel for subset of Java legacy programs

being invoked by others.

- Method definition: A method M definition of a class corresponds to a sub-machine connected to the new object state with an incoming and an outgoing transitions from and to the new object, respectively The incoming transition is labeled with an even M(). The outgoing transition notifies the invoking state machine with a M_done message.

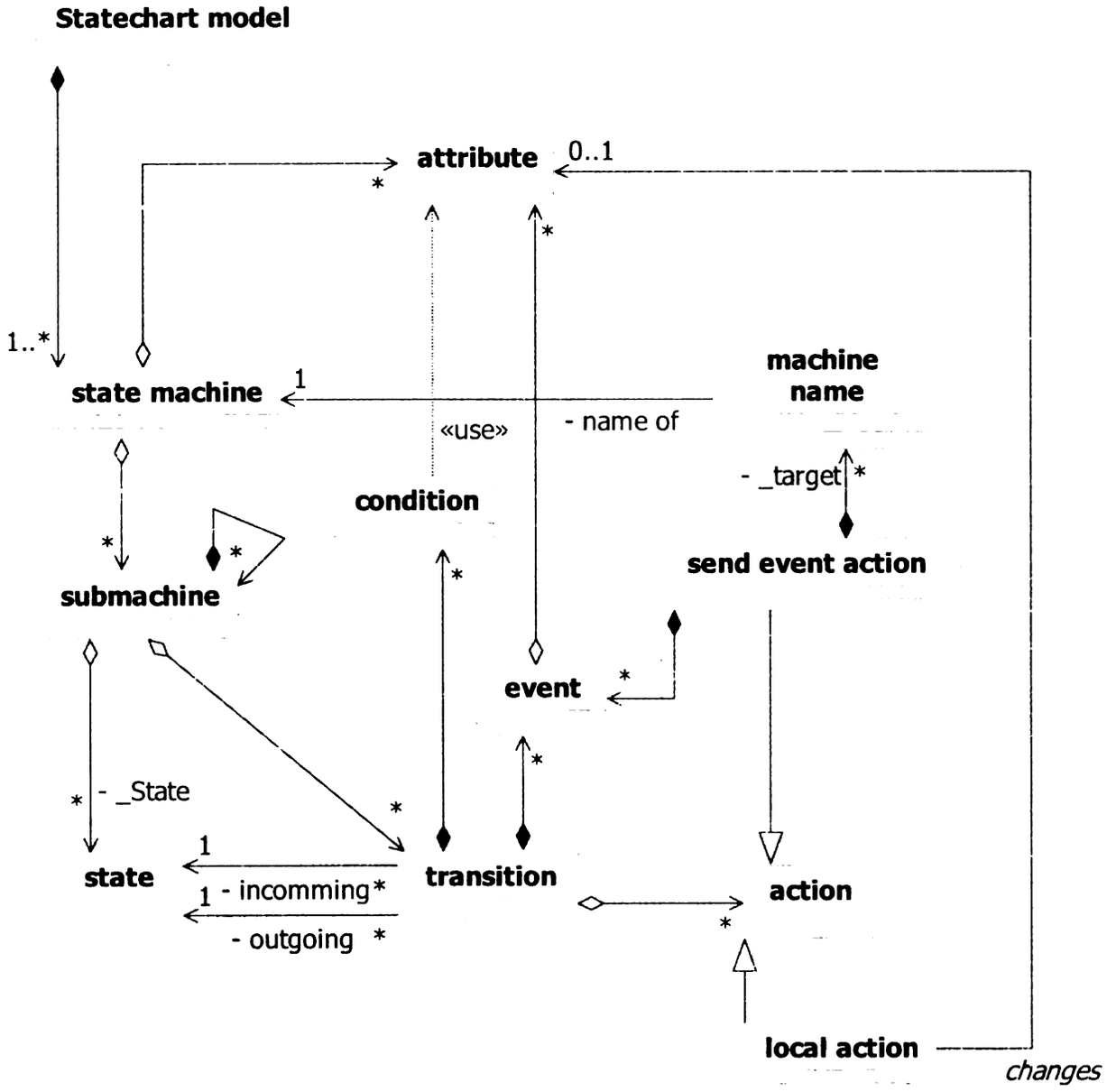
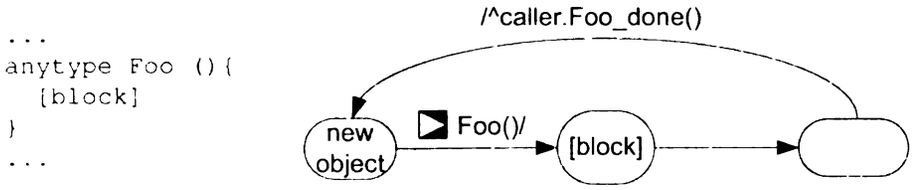


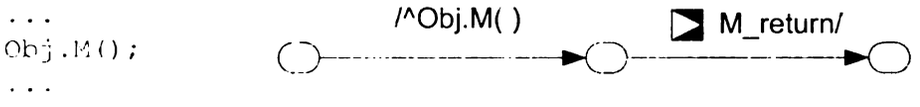
Figure C.2: Metamodel for subset of UML Statechart diagrams



(a) Java feature (b) Statechart feature

Method definition

- Method invocation: The invocation of a method M of an object Obj is modeled as a transition that sends a message M to the object Obj.
- Method return: The return from a method invocation M is modeled as a transition labeled with an event M_return.



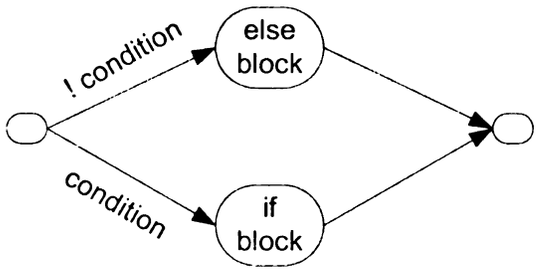
(a) Java feature (b) Statechart feature

Method invocation and return

- Attributes: Attributes of a Java class correspond to the attributes of the state machine for the class.
- Conditions: A condition in Java is a logic expression evaluated over attributes. They correspond to conditions in the Statechart diagram.
- Simple block: A simple block is a Java statement that only accesses local attributes. A simple block is modeled as a transition with local actions, i.e., actions that only accesses local attributes.
- If-else block: An if-else block corresponds to a branching structure in the state machine, where the two branches correspond to the if and else blocks, respec-

tively.

```
...
if (condition){
  [if block]
}else {
  [else block]
}
...
```

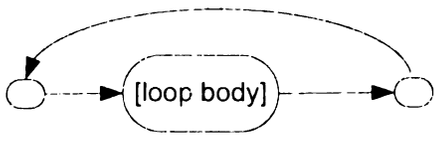


(a) Java feature (b) Statechart feature

If-else block

- Loop block: A loop block corresponds to the state machine for the loop body and a transition going back to the entry state.

```
...
while(true){
  [loop body]
}
...
```

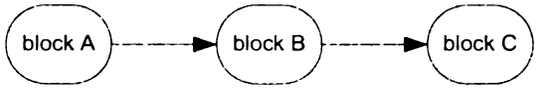


(a) Java feature (b) Statechart feature

Loop block

- Sequential block: A sequential block contains a sequence of blocks. It is modeled as a sequence of state machines connected by transitions.

```
...
[block A]
[block B]
[block C]
...
```



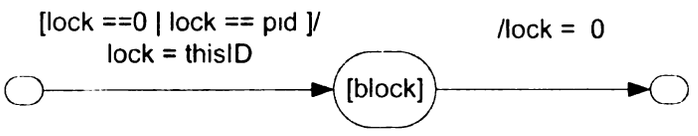
(a) Java feature (b) Statechart feature

Sequential block

- Synchronized block: At the entry point of a synchronized block, the program must acquire the monitor for the object. At the exit point of a synchronized

block, the program must release the monitor. These are modeled as transitions with actions setting and resetting the value of the attribute lock.

```
...
synchronized
(lock){
    [block]
}
...
```



(a) Java feature (b) Statechart feature

Synchronized block

Bibliography

[1] Vikram Adve, Vinh Vi Lam, and Brian Ensink. Language and compiler support for adaptive distributed applications. In *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001)*, Snowbird, Utah, June 2001.

[2] Robert Allen, Remi Douence, and David 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.

[3] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–203, 1994.

[4] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst*, 23(3):273–303, 2001.

[5] J. Appavoo, K. Hui, C. A. N. Soules, et al. Enabling autonomic behavior in systems software with hot swapping. *IBM System Journal*, 42(1):60, 2003.

[6] Jonathan Appavoo, Robert W. Wisniewski, Craig A. N. Soules, et al. An infrastructure for multiprocessor run-time adaptation. In *Proceedings of the ACM SIGSOFT' Workshop on Self-Healing Systems (WOSS02)*, November 2002.

[7] Anish Arora and Mohamed G. Gouda. Distributed reset. *IEEE Trans. Comput.*, 43(9):1026–1038, 1994.

[8] H.P. Barendregt. *The Lambda calculus: Its Syntax and Semantics*. North Holland, 2nd reprint 1997 edition edition, November 2006.

[9] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with ltl in eagle. In *18th International Parallel and Distributed Processing Symposium, Parallel and Distributed Systems: Testing and Debugging - PADTAD'04*. IEEE Computer Society Press, April 2004. ISBN 0769521320.

[10] Daniel M. Berry, Betty H. C. Cheng, and Ji 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 (REFSQ'05)*, Porto, Portugal, June 2005.

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283. ACM Press, 1995.

[12] Brian N. Bershad et al. Spin - an extensible microkernel for application-specific operating system services. Technical report, Dept. of Computer Science and Engineering, University of Washington, 1994.

[13] P. Bertrand, R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: an environment for goal drivent requirements engineering. In *Proceedings of the 20th International Conference on Software Engineering*. IEEE Computer Society, 1998.

[14] Karun Biyani and Sandeep Kulkarni. Mixed-mode adaptation in distributed systems: A case study. In *Proceedings of International Workshop on Software Engineering for Adaptive and Self-Managing Systems - SEAMS, at ICSE*, May 2007.

[15] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. Springer-Verlag.

[16] Jean-Chrysotome Bolot and Andres Vega-Garcia. Control mechanisms for packet audio in Internet. In *Proceedings of IEEE INFOCOM'96*, pages 232–239, San Francisco, California, April 1996.

[17] Howard Bowman and Simon J. Thompson. A tableaux method for Interval Temporal Logic with projection. In *TABLEAUX'98, International Conference on Analytic Tableaux and Related Methods*, number 1397 in Lecture Notes in AI pages 108–123. Springer-Verlag, May 1998.

[18] J.S. Bradbury, J.R. 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)*, pages 28–33. Newport Beach, California, October/November 2004.

[19] Carlos Canal, Ernesto Pimentel, and José M. Troya. Specification and refinement of dynamic software architectures. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*. pages 107–126. Kluwer, B.V., 1999.

[20] A. Cau, B. Moszkowski, and H. Zedan. Interval temporal logic. Internet, Sep 2002.

[21] Feng Chen, Marcelo d'Amorim, and Grigore Rosu. Checking and correcting behaviors of java programs at runtime with java-mop. *Electr. Notes Theor. Comput. Sci.*, 144(4):3–20, 2006.

[22] Wen-Ke Chen, Matti A. Hiltunen, and Richard 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.

[23] Betty H. C. Cheng, Laura A. Campbell, Min Deng, and R.E.K. Stirewalt. Enabling validation of uml formalizations. Technical Report MSU-CSE-02-25, Department of Computer Science, Mich State Univ, East Lansing, MI, September 2002.

[24] D. M. Chess, C. Palmer, and S. R. White. Security in an autonomic computing environment. *IBM System Journal*, 42(1):107–118, 2003.

[25] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. T orielli, and P. Traverso. Model checking safety-critical software with SPIN: An application to a railway interlocking system. In *Proc. of the 17th Int. Conf. Computing Safety. Reliability. and Security*, pages 284–295, Heidelberg, October 1998.

[26] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. FLAVERS: A finite state verification technique for software systems. *IBM Systems Journal*, 41(1):140–165, March 2002.

[27] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 97–108, New York, NY, USA, 2006. ACM Press.

[28] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[29] C. Courcoubetis, M. Vardi, P Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2-3):275–288, 1992.

[30] Costas Courcoubetis. Minimum and maximum delay problems in real-time systems. In *CAV '91: Proceedings of the 3rd International Workshop on Computer Aided Verification*, pages 399–409, London, UK, 1992. Springer-Verlag.

[31] K. Narayanaswamy D. Cohen, M. S. Feather and S. Fickas. Automatic monitoring of software requirements. In *Proc. 19th International Conference on Software Engineering*, Boston, 1997.

[32] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[33] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 21–26. ACM Press, 2002.

[34] Jutta Degener and Carsten Bormann. The GSM 06.10 lossy speech compression library and its applications, 2000. available at http://kbs.cs.tu-berlin.de/ jutta/toast.html.

[35] Doron Drusinsky. The temporal rover and the atg rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, 2000. Springer-Verlag.

[36] Matthew B. Dwyer and Corina S. Păsăreanu. 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*, pages 189-202. ACM Press, 1998.

[37] E. Allen Emerson. Temporal and modal logic. *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 995-1072, 1990.

[38] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 169-180. ACM Press, 1982.

[39] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251-266, 1995.

[40] ETSI. European Telecommunications Standards Institute. URL: `http://www.etsi.org/`.

[41] M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *Proceedings of the 9th International Workshop on Software Specification and Design*, page 50. IEEE Computer Society, 1998.

[42] Martin Feather. Using SPIN to analyze properties of flight systems. Private Communication.

[43] Steve Fickas, Gerd Kortuem, and Zary Segall. Software organization for dynamic and adaptable wearable systems. In *Proceedings First International Symposium on Wearable Computers (ISWC'97)*, 1997.

[44] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference held jointly with the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 152-163, New York, NY, USA, 2001. ACM Press.

[45] Kathi Fisler and Shriram Krishnamurthi. Decomposing verification by features. In *IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, 2006.

[46] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *SPIN 03: SPIN Workshop, LNCS 2648*, pages 213-225. Springer-Verlag, 2003.

[47] Scott D. Fleming, Betty H. C. Cheng, R. E. Kurt Stirewalt, and Philip K. McKinley. An approach to implementing dynamic adaptation in c++. *SIGSOFT Softw. Eng. Notes*, 30(4):1-7, 2005.

[48] J. Froessl, Th. Kropf, and J. Gerlach. An efficient algorithm for real-time symbolic model checking. In *EDTC '96: Proceedings of the 1996 European conference on Design and Test*, page 15, Washington, DC, USA, 1996. IEEE Computer Society.

[49] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, January 1995.

[50] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM System Journal*, 42(1):5, 2003.

[51] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification (PSTV95)*, Warsaw, Poland, June 1995.

[52] Rodolfo Gomez and Howard Bowman. PITL2MONA: Implementing a decision procedure for propositional interval temporal logic. *Journal of Applied Non-Classical Logics*, 14(1-2):105--148, 2004.

[53] Jim Gray Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393-481. Springer-Verlag, 1978.

[54] David Harel, Dexter Kozen, and Rohit Parikh. Process logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144 170, 1982.

[55] K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. In *Proceedings of the 1st Workshop on Runtime Verification*, Paris, France, July 2001.

[56] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A.A. Sanvido. Extreme model checking. *Verification: Theory and Practice, Lecture Notes in Computer Science 2772, Springer-Verlag*, pages 332-358, 2004.

[57] Dan Hirsch, Paolo Inverardi, and Ugo Montanari. Graph grammars and constraint solving for software architecture styles. In *Proceedings of the third international workshop on Software architecture*, pages 69-72. ACM Press, 1998.

[58] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, NJ, USA, 1985.

[59] Gerald J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), may 1997.

[60] Gerald J. Holzmann. Software verification at bell labs: One line of development. *Bell Labs Technical Journal*, pages 35 45, January March 2000.

[61] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[62] IBM. Ibm software - rational rose xde developer for java - product overview. online.

[63] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

[64] Bengt Jonsson and Yih-Kuen Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(1-2):47–72, 1996.

[65] E. Kasten, P. K. McKinley, S. Sadjadi, and R. Stirewalt. Separating introspection and intercession in metamorphic distributed systems. In *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02)*, 2002,KMSS02.AOPDCS.

[66] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[67] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[68] Ralf Kollmann, Petri Selonen, Eleni Stroulia, Tarja Syst à, and Albert Zü ndorf. A study on the current state of the art in tool-supported uml-based static reverse engineering. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 22–32, 2002.

[69] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Claudio Magalhães;, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *IFIP/ACM International Conference on Distributed systems platforms*, pages 121–143. Springer-Verlag New York, Inc., 2000.

[70] Ron Koymans. *Specifying message passing and time-critical systems with temporal logic. LNCS 651.* Springer-Verlag. 1992.

[71] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[72] Jeff Kramer and Jeff Magee. Analysing dynamic change in software architectures: a case study. In *Proc. of 4th IEEE International Conference on Configuratble Distributed Systems*, Annapolis, May 1998.

[73] Jeff Kramer, Jeff Magee, and Morris Sloman. Configuring distributed systems. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop*, pages 1–5. ACM Press, 1992.

[74] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 137–146, New York, NY, USA, 2004. ACM Press.

[75] L.M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer, Springer Verlag*, pages 98–132, 1998.

[76] Sandeep Kulkarni and Karun Biyani. Correctness of component-based adaptation. In *Proceedings of International Symposium on Component-based Software Engineering*, May 2004.

[77] Sandeep S. Kulkarni, K. N. Biyani, and U. Arumugam. Composing distributed fault-tolerance components. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN), Supplemental Volume, Workshop on Principles of Dependable Systems*, pages W127–W136, June 2003.

[78] Olaf Kummer and Frank Wienberg. Renew - the reference net workshop. In *In Tool Demonstrations, 21st International Conference on Application and Theory of Petri Nets*, pages 28–30, Aarhus, Denmark, 2000.

[79] Olaf Kummer, Frank Wienberg, and Michael Duvigneau. *Renew–User Guide*. University of Hamburg, Department for Informatics, Theoretical Foundations Group, Distributed Systems Group, Hamburg German, release 2.0.1 edition, October 2004.

[80] Orna Kupferman and Moshe Y. Vardi. Modular model checking. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 381–401, London, UK, 1998. Springer-Verlag.

[81] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[82] Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Yijun 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.

[83] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime assurance based on formal specifications. In *Proc. Parallel and Distributed Processing Techniques and Applications*, pages 279–287, 1999.

[84] Emmanuel Letier. *Reasoning about Agents in Goal-Oriented Requirements Engineering.* PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 2001.

[85] Y. Levendel. Delivering dependable telecommunication services using off-the-shelf system components. *J. Electron. Test.*, 12(1-2):153–159, 1998.

[86] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying cross-cutting features as open systems. *ACM SIGSOFT Software Engineering Notes*, 27(6):89–98, 2002.

[87] Jingyue Li, Reidar Conradi, Odd Petter N. Slyngstad, Christian Bunse, Marco Torchiano, and Maurizio Morisio. An empirical study on decision making in off-the-shelf component-based development. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 897–900, New York, NY, USA, 2006. ACM Press.

[88] Orna Lichtenstein and Amir 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*, pages 97–107. ACM Press, 1985.

[89] G. Logothetis and K. Schneider. A new approach to the specification and verification of real-time systems. In *Euromicro Conference on Real Time Systems*, pages 171–180, Delft, The Netherlands, June 2001. IEEE Computer Society.

[90] Pattie Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155. ACM Press, 1987.

[91] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[92] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proc. of the 4th Symposium on the Foundations of Software Engineering (FSE4)*, page 314, San Francisco,CA, October 1996.

[93] Jeff Magee. Behavioral analysis of software architectures using ltsa. In *Proceedings of the 21st International Conference on Software Engineering*, pages 634–637. IEEE Computer Society Press, 1999.

[94] Marko 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*, pages 434–444, London, UK, 2002. Springer-Verlag.

[95] Philip K. McKinley. RAPIDware. http://www.cse.msu.edu/rapidware/. Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan.

[96] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[97] William E. McUmber and Betty H. C. Cheng. A general framework for formalizing uml with formal languages. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 433–442, Washington, DC, USA, 2001. IEEE Computer Society.

[98] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 60–76, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[99] Daniel Le Métayer. Software architecture styles as graph grammars. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 15–23. ACM Press, 1996.

[100] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[101] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[102] Ben Moszkowski. A hierarchical analysis of propositional temporal logic based on intervals. In Sergei Artemov, Howard Barringer, Artur S. d'Avila Garcez, Luis C. Lamb, and John Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 2, pages 371–440. College Publications, 2005.

[103] M.Y.Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st Symposium on Logic in Computer Science*, pages 322–331, Cambridge, England, 1986.

[104] Jeff Offutt, Mary Jean Harrold, and Priyadarshan Kolte. A software metric system for module coupling. *The Journal of Systems and Software,Elsevier*, 20(3):295–308, March 1993.

[105] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 1999.

[106] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, pages 177–186. IEEE Computer Society, 1998.

[107] Jonathan S. Ostroff. Composition and refinement of discrete real-time systems. *ACM Trans. Softw. Eng. Methodol.*, 8(1):1–48, 1999.

[108] David L. Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976.

[109] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.

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

[111] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[112] Alexander Pretschner. Model-based testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 722–723, New York, NY, USA, 2005. ACM Press.

[113] R. Razouk and M. Gorlick. Real-time interval logic for reasoning about executions of real-time programs. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 10–19, New York, NY, USA, 1989. ACM Press.

[114] Barry Redmond and Vinny Cahill. Iguana/j: Towards a dynamic and efficient reflective architecture for java. In *workshop on Reflection and Meta-Level Architectures at 14th European Conference on Object-Oriented Programming*, Cannes,France, June 2000.

[115] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002.

[116] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, April 1997.

[117] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267–276, New York, NY, USA, 2003. ACM Press.

[118] Jonathan Rosenberg, Lili Qiu, and Henning Schulzrinne. Integrating packet FEC into adaptive voice playout buffer algorithms on the Internet. In *Proceedings of IEEE INFOCOM 2000*, pages 1705–1714, 2000.

[119] R. Rosner and A. Pnueli. A choppy logic. In *1st IEEE Symposium on Logic in Computer Science*, pages 306–313, 1986.

[120] S. M. Sadjadi, P. K. McKinley, and E. P. Kasten. Architecture and operation of an adaptable communication substrate. In *Proceedings of the Ninth IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, pages 46–55, San Juan, Puerto Rico, May 2003.

[121] S. Masoud Sadjadi and Philip K. McKinley. Using transparent shaping and web services to support self-management of composite systems. In *Proceedings of the Second IEEE International Conference on Autonomic Computing (ICAC)*, Seattle, Washington, June 2005.

[122] S. Masoud Sadjadi, Philip K. McKinley, Betty H. C. Cheng, and R.E. Kurt Stirewalt. TRAP/J: Transparent generation of adaptable java programs. In *Proceedings of the International Symposium on Distributed Objects and Applications*, Agia Napa, Cyprus, October 2004.

[123] F. Schneider, S. Easterbrook, J. Callahan, and G. Holzmann. Validating requirements for fault tolerant systems using model checking. In *Proc. of the 3rd International Conference on Requirements Engineering*, pages 4–13, Colorado Springs, Colorado, April 1998.

[124] A. P. Sistla. On characterization of safety and liveness properties in temporal logic. In *PODC '85: Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, pages 39–48. ACM Press, 1985.

[125] Asim Smailagic, Daniel P. Siewiorek, Richard Martin, and John Stivoric. Very rapid prototyping of wearable computers: a case study of custom versus off-the-shelf design methodologies. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 315–320, New York, NY, USA, 1997. ACM Press.

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

[127] Peter Norvig Stuart J. Russell. *Artificial Intelligence: A Modern Approach*, chapter 9 Inference in First-Order Logic, page 274. Prentice Hall, second edition, 2002.

[128] Gabriele Taentzer, Michael Goedicke, and Torsten 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*, pages 179–193. Springer-Verlag, 2000.

[129] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 4 edition, Aug. 2002.

[130] Chiping Tang and Philip K. McKinley. Improving multipath reliability in topology-aware overlay networks. In *Proceedings of the Fourth International Workshop on Assurance in Distributed Systems and Networks*, Columbus, Ohio, June 2005.

[131] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., and Jason E. Robbins. A component- and message-based architectural style for GUI software. In *Proceedings of the 17th International Conference on Software Engineering*, pages 295–304. ACM Press, 1995.

[132] The AspectJ Team. The aspectj(tm) programming guide.

[133] Wolfgang Thomas. Automata on infinite objects. *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 133–191, 1990.

[134] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, page 249. IEEE Computer Society, 2001.

[135] Michael VanHilst and David Notkin. Using C++ Templates to Implement Role-Based Designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer Verlag, 1996.

[136] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

[137] Thomas Wilke. Classifying discrete temporal properties. In Christoph Meinel, editor, *STACS'99*, volume 1563 of *Lecture Notes in Computer Science*, pages 32–46, Trier, Germany, 1999. Springer.

[138] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proceedings of 24th Symposium on Foundations of Computer Science*, pages 185–194. IEEE Computer Society, Nov 1983.

[139] Zhenxiao Yang, Betty H. C. Cheng, Kurt Stirewalt, Masoud Sadjadis, Jesse Sowell, and Philip McKinley. An aspect-oriented approach to dynamic adaptation. In *Proceedings of the ACM SIGSOFT Workshop on Self-Healing Systems*, 2002.

[140] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Department of Computer Science, 1995.

[141] Y. Yu, J. Mylopoulos, A. Lapouchnian, S. Liaskos, , and J. C. S. P. Leite. From stakeholder goals to high-variability software designs. *Technical Report CSRG-509, University of Toronto*, 2005.

[142] Y. Yu, Y. Wang, S. Easterbrook, A. Lapouchnian, S. Liaskos, and J. Leite. Configuring common personal software: a requirements-driven approach. Technical Report Technical Report CSRG-512, University of Toronto, 2005.

[143] Ji Zhang and Betty H. C. Cheng. Specifying adaptation semantics. In *WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1–7, St. Louis, Missouri, May 2005. ACM Press.

[144] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of International Conference on Software Engineering (ICSE'06)*, Shanghai,China, May 2006.

[145] Ji Zhang and Betty H. C. Cheng. Modular model checking of dynamically adaptive programs. Technical Report MSU-CSE-06-18, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, March 2006. `http://www.cse.msu.edu/~zhangji9/Zhang06Modular.pdf`.

[146] Ji Zhang and Betty H. C. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software (JSS), Architecting Dependable Systems*, 79(10):1361–1369, 2006.

[147] Ji Zhang and Betty H. C. Cheng. Amoeba-runtime: Run-time verification of adaptive software. Technical Report MSU-CSE-07-16, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, March 2007.

[148] Ji Zhang and Betty H. C. Cheng. Re-engineering legacy systems for assured dynamic adaptation. Technical Report MSU-CSE-07-11, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, February 2007.

[149] Ji Zhang and Betty H. C. Cheng. Towards re-engineering legacy systems for assured dynamic adaptation. In *Workshop on Modeling in Software Engineering (accepted for publication)*, Minneapolis, Minnesota, May 2007.

[150] Ji Zhang, Betty H. C. Cheng, Zhenxiao Yang, and Philip K. McKinley. Enabling safe dynamic component-based software adaptation. *Architecting Dependable Systems. Lecture Notes in Computer Science*, pages 194–211, 2005.

[151] Ji Zhang, Jaejin Lee, and Philip K. McKinley. Optimizing the Java pipe I/O stream library for performance. In *Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Published as Lecture Notes in Computer Science (LNCS), Vol. 2481, Springer-Verlag*, College Park, Maryland, USA, July 2002.

[152] Ji Zhang, Zhenxiao Yang, Betty H. C. Cheng, and Philip K. McKinley. Adding safeness to dynamic adaptation techniques. In *Proceedings of ICSE 2004 Workshop on Architecting Dependable Systems*, Edinburgh, Scotland, UK, May 2004.

[153] Ji Zhang, Zhinan Zhou, Betty H. C. Cheng, and Philip K. McKinley. Specifying real-time properties in autonomic systems. *Journal Innovations in Systems and Software Engineering*, January 2007.

[154] Zhinan Zhou, Philip 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.

[155] Zhinan Zhou, Ji Zhang, Philip K. McKinley, and Betty H. C. Cheng. TA-LTL: Specifying adaptation timing properties in autonomic systems. In *3rd IEEE Workshop on Engineering of Autonomic Systems (EASe 2006)*, Columbia, Maryland, April 2006.