Adding Safeness to Dynamic Adaptation Techniques
(Work-in-Progress)*

Ji Zhang, Zhenxiao Yang, Betty H.C. Cheng, and Philip K. McKinley
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
Email: {zhangji9,yangzhe1,chengb,mckinley}@cse.msu.edu

Abstract

Recomposable software enables a system to change its structure and behavior during execution, in response to a dynamic execution environment. This paper proposes an approach to ensure that such adaptations are safe with respect to system consistency. The proposed method takes into consideration dependency analysis for target components, specifically determining viable sequences of adaptive actions and those states in which an adaptive action may be applied safely. We demonstrate the technique that ensures safe adaptation (insertion, removal, and replacement of components) in response to changing external conditions in a wireless multicast video application.

Keywords: safe adaptation, dynamic adaptation, reliability, autonomic computing, middleware, component-based software, dependency analysis

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. The need for adaptability is perhaps most acute at the “wireless edge” of the Internet, where mobile devices balance several conflicting and possibly cross-cutting concerns, including quality of service on wireless connections, changing security policies, and energy consumption.

To meet the needs of emerging and future adaptive systems, numerous research efforts in the past several years have addressed ways to construct adaptable software. Examples include support for adaptability in programming languages [1–3], frameworks to design context-aware applications [4, 5], adaptive middleware platforms that shield applications from external dynamics [6, 7], and adaptable and extensible operating systems [8].

Despite these advances in mechanisms used to build recomposable software, the full potential of dynamically recomposable software systems can be realized only if the adaptation is performed in a disciplined manner. We use the term “safe adaptation” to mean the program maintains its integrity during adaptation. An adaptation is safe if (1) it does not violate dependency relationships and (2) it does not interrupt communication either within a component or between components that would potentially yield erroneous or unexpected results. For discussion purposes, we use the term atomic communication to refer to the communication scenarios mentioned in the second part of the definition. Unless adaptive software mechanisms are grounded in formalisms that codify invariants and other properties that must hold during recomposition, the resulting systems will be prone to errant behavior.

This paper describes work-in-progress on safe adaptation in dynamically recomposable systems. This work is part of an ONR-sponsored project called RAPIDware that addresses the design of adaptive middleware for dynamic, heterogeneous environments. Such systems require run-time adaptation, including the ability to modify and replace components, in order to survive hardware component failures, network outages, and security attacks.

Dynamically adaptive software development comprises four major tasks: Enabling adaptation makes a program adapt-ready, i.e., capable of run-time reconfiguration. Program monitoring instruments the program and monitors condition changes in the execution environment. Decision-making determines when and how the program should be modified. Process management ensures the safe adaptation.

* This work has been supported in part by NSF grants EIA-0000433, EIA-0130724, ITR-0313142, and CCR-9901017, and the Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744.
Our previous work has focused primarily on developing techniques for the first three tasks. We developed Adaptive Java [2], an extension to Java that supports dynamic reconfiguration of software components. We have used Adaptive Java to construct a variety of adaptive components, including MetaSockets [9]. MetaSockets are created from existing Java socket classes, but their structure and behavior can be adapted at run time in response to external stimuli. Specifically, MetaSocket behavior can be adapted through the insertion and removal of filters that manipulate the passing data stream. For example, filters can perform encryption, decryption, forward error correction, compression, and so forth. In order to maintain a separation of concerns between the original program and the code responsible for adaptation, we applied an aspect-oriented approach to dynamic adaptation [10], where we used MetaSockets to make programs adapt-ready for adaptations at run time.

This paper focuses on the fourth task, specifically, ensuring that a dynamically adaptive action is performed safely. Adaptive actions can involve the insertion of a new component, removal of a component, or the replacement of an existing component. Our approach to ensuring safeness during adaptation offers three advantages to other approaches [11–14]. First, we use invariants to specify dependency relationships among multiple components executing across a single or distributed processes. These dependency relationships enable us to determine which components are affected during a given adaptation, and consequently the set of safe states in which dynamic adaptations can take place. Second, our approach provides centralized management of adaptations, thus enabling optimizations to be made when more than one set of adaptive actions can be used to satisfy a given adaptation need. Third, our approach provides a rollback mechanism in case an error or failure is encountered during the adaptation process.

We have applied our safeness techniques to adaptive applications developed with MetaSockets and Adaptive Java, primarily in the mobile computing domain. The remainder of this paper is organized as follows. Section 2 describes our proposed approach to safe adaptation, and Section 3 describes its use in a video multicasting application using MetaSockets. Section 4 discusses related work, and Section 5 concludes the paper.

2. Safe Adaptation Process

A component-based software system can be modeled as a set of communicating components running on one or more processes. The communication among components refers to any type of interaction, including message exchange, function calls, IPC, RPC, network communication, and so on. The adaptations we consider are component insertion, removal, replacement, and combinations thereof. Unsafe adaptation often originates when one of these actions produces a dysfunctional communication pattern among components. Dependencies between a candidate adaptive component and components with which it communicates must be assessed to determine when it is safe to perform an adaptive action. Global safe states are those in which no atomic communication segments are interrupted with respect to a given adaptive action. For discussion purposes, we use the term atomic communication to refer to an interaction (communication) either within a component or between components that cannot be interrupted, otherwise, it would potentially yield erroneous or unexpected results. Local safe states are those in which no atomic communication is interrupted on a local process during an adaptation. For example, if the adaptive action is to change an encryption filter, then the dependencies between the processing performed by the encryption filter and the processing by the decryption filter must be established to determine when it is safe to change the filters.

Our proposed method is executed by a safe adaptation manager, typically a separate process that is responsible for adaptations of the entire system. The manager communicates with adaptation agents attached to processes. An agent receives commands from the manager, performs adaptive actions, and reports the status of the local process to the manager. Communication channels can be implemented to best match the particular system. For example, both Arora [15] and Kulkarni [13] have used a spanning tree, which is well suited to hierarchical structures. In contrast, in a group communication system, multicast may be a better mechanism for coordination between the manager and the agents for the processes.

Our approach comprises three major steps. These steps are achieved by the coordination of the manager and the agents at run time after the adaptation decision is made and the target configuration is sent to the manager.

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 vertices and adaptation steps as arcs. Here we assume that a given adaptive action may be decomposed into several adaptation steps. For example, if the adaptive action is to change the encryption filter, then the intermediate adaptive steps include changing the encryption encoder, changing the decryption decoder, monitoring the message buffer to determine when it is “safe” to swap the encoder or decoder, blocking the transmission of data, etc.

3. Find Minimum Safe Adaptation Path (MAP). Finally, we apply Dijkstra’s shortest path algorithm on
the SAG to find a feasible solution with minimum weight, where fixed costs have been associated with each adaptation step. Factors affecting cost include system blocking time, adaptation duration, delay of packet delivery, resource usage, etc.

Managing the adaptations. Once the system detects a condition warranting adaptation, the manager and the agents coordinate to achieve the adaptation. First, the manager analyzes the dependency relationships relative to the requested adaptation and prepares a MAP. Second, after selecting an adaptive path from the MAP, the manager sends a reset message to the agents. After receiving a reset done message from all agents, then all the affected processes are in local safe states, and the overall system is in a global safe state. At this point, the process involving adaptation may proceed. Once all of the agents of the adapting processes send an adaptation done message back to the manager, the manager sends a resume message to the agents. Third, once all the acknowledgements for the resume message have been received from the agents, then the manager proceeds to process any remaining adaptation steps in the adaptation path or resume full operation of the system.

We use a timeout mechanism to detect failures during the adaptation process. Failures are unexpected conditions or events that affect one or more steps of the adaptation process (e.g., lack of availability of component). In the event that a failure occurs during the adaptation process, several options are possible depending on how far the adaptation process has reached: termination of all adaptive actions, roll back all affected processes to the state prior to adaptation, or retrying the adaptive action.

3. Video Streaming Example

We use a (relatively simple) video multicasting system to illustrate the safe adaptation process. Figure 1 shows the configuration of the application, comprising a video server and one or more video clients. In this example, one client is a hand-held computer with a short battery life and limited computing power, and the second client is a laptop 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 decomposed into video frames. Finally the frames are displayed in a video player.

In this example, two main encryption schemes are available for processing the data: DES 64-bit encoding/decoding, and DES 128-bit encoding/decoding.

Figure 1: Configuration of the video streaming application

The sender has two components: E1, a DES 64-bit encoder and E2, a DES 128-bit encoder. 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 the 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 manager and attach an agent thread to both the server and the clients, respectively. In this particular application and system architecture, the manager uses a direct TCP connection to communicate with the agents.

Safe Adaptation Path and MAP. 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. So we have the following invariants:

- System Invariants:
  - Resource Constraint: (D1, D2, D3)
    One of the receivers, the hand-held device, allows only one DES decoder to be in the system at a given time due to computing power constraints.
  - Security constraint: (E1, E2):
    The sender should have one encoder in the system so that the data is encoded during the adaptation.

Where \(\oplus\) represents “exclusively select one from a given set of elements”.

- Dependency invariants
  - \(E1 \rightarrow (D1 \lor D2) \land D4\)
    The combination of decoders needed by E1-encoded packets.
− $E2 \rightarrow (D3 \lor D2) \land D5$

The combination of decoders needed by E2-encoded packets.

Where $\rightarrow$ represents the dependency relationship; For example, $A \rightarrow B$ means that the correctness of $A$ depends on the correctness of $B$.

We input source and target configurations to the manager, which uses the dependency relationship expressions to generate 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. The source configuration is $(0100101)$ and the target configuration is $(1010010)$.

The resulting safe configuration set is shown in Table 1. The adaptive actions shown in Table 2 are input to the manager. Only related actions are listed. The cost column is packet delay in milliseconds. Note, 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 have much higher cost values than other actions.

<table>
<thead>
<tr>
<th>Configuration No.</th>
<th>Safe configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>0100101</td>
</tr>
<tr>
<td>5-8</td>
<td>1110010 1100101 1101001 1101010</td>
</tr>
</tbody>
</table>

Table 1. SCS: Safe Configuration Set

<table>
<thead>
<tr>
<th>Action</th>
<th>Operation</th>
<th>Cost (ms)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>$E1 \rightarrow E2$</td>
<td>10</td>
<td>replace $E1$ with $E2$</td>
</tr>
<tr>
<td>A2</td>
<td>$D1 \rightarrow D2$</td>
<td>10</td>
<td>replace $D1$ with $D2$</td>
</tr>
<tr>
<td>A3</td>
<td>$D1 \rightarrow D3$</td>
<td>10</td>
<td>replace $D1$ with $D3$</td>
</tr>
<tr>
<td>A4</td>
<td>$D2 \rightarrow D3$</td>
<td>10</td>
<td>replace $D2$ with $D3$</td>
</tr>
<tr>
<td>A5</td>
<td>$D1 \rightarrow D6$</td>
<td>10</td>
<td>replace $D4$ with $D5$</td>
</tr>
<tr>
<td>A6</td>
<td>$(D1, E1) \rightarrow (D2, E2)$</td>
<td>100</td>
<td>$(A1 \land A2)$</td>
</tr>
<tr>
<td>A7</td>
<td>$(D1, E1) \rightarrow (D5, E2)$</td>
<td>100</td>
<td>$(A1 \land A3)$</td>
</tr>
<tr>
<td>A8</td>
<td>$(D2, E1) \rightarrow (D5, E2)$</td>
<td>100</td>
<td>$(A1 \land A4)$</td>
</tr>
<tr>
<td>A9</td>
<td>$(D4, E1) \rightarrow (D5, E2)$</td>
<td>100</td>
<td>$(A1 \land A5)$</td>
</tr>
<tr>
<td>A10</td>
<td>$(D1, D4) \rightarrow (D2, D5)$</td>
<td>50</td>
<td>$(A2 \land A3)$</td>
</tr>
<tr>
<td>A11</td>
<td>$(D1, D4) \rightarrow (D3, D5)$</td>
<td>50</td>
<td>$(A2 \land A5)$</td>
</tr>
<tr>
<td>A12</td>
<td>$(D2, D4) \rightarrow (D3, D5)$</td>
<td>50</td>
<td>$(A4 \land A5)$</td>
</tr>
<tr>
<td>A13</td>
<td>$(D5, D4, E1) \rightarrow (D2, D5, E2)$</td>
<td>150</td>
<td>$(A1 \land A10)$</td>
</tr>
<tr>
<td>A14</td>
<td>$(D1, D4, E1) \rightarrow (D5, D5, E2)$</td>
<td>150</td>
<td>$(A1 \land A11)$</td>
</tr>
<tr>
<td>A15</td>
<td>$(D2, D4, E1) \rightarrow (D3, D5, E2)$</td>
<td>150</td>
<td>$(A1 \land A12)$</td>
</tr>
<tr>
<td>A16</td>
<td>$-D4$</td>
<td>10</td>
<td>remove $D4$</td>
</tr>
<tr>
<td>A17</td>
<td>$+D5$</td>
<td>10</td>
<td>insert $D5$</td>
</tr>
</tbody>
</table>

Table 2. Adaptive Actions and Cost.
Magee [12] have also developed a dynamic change management method that separates management of structural aspects of the system from component application concerns, thus enabling the system to evolve during a quiescent state. Kulkarni et al. [13] propose an approach to safely composing distributed fault-tolerance components at run time. In their work, they use a spanning tree of the system processes to pass messages between the root (the initiator of the recomposition) and other processes to synchronize the recomposition process. Analyzing two components at a time, their work identifies the possible dependency relationships among fractions of a distributed component. They also use a reset mechanism [15] to block computations during the recomposition process. Amano et al. [14] introduced a model for flexible and safe mobile code adaptation, where adaptations are serialized if there are dependencies among adapted components. Their approach supports the use of assertions for specifying preconditions and postconditions for adaptation, where violations will cancel the adaptation or rollback the adaptation, respectively. They use object invariants to specify the methods and the order of method invocations for the adaptive procedures.

5. Conclusions

This paper presents an approach to achieve safeness during dynamic adaptation that can be used in conjunction with existing dynamic adaptation techniques. We use a centralized adaptation manager to schedule the adaptation process, which results in a globally minimum solution. We block the newly added 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 atomicity of adaptive actions. The process we introduce is largely automated: The algorithms are carried out by the manager and the agent programs. As with other approaches that use dependency relationships, the developers specify the dependency relationships and the adaptive actions.

While MAP is the goal of the path searching algorithm, we are investigating an approximation algorithm to perform path searching that has complexity $(mn^2)$ where $n$ is the number of adaptive components involved and $m$ is the number of eligible adaptive actions.

References


