Building Component Families to Support Adaptation *

Karun N. Biyani Sandeep S. Kulkarni
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824 USA
{biyanika,sandeep}@cse.msu.edu

ABSTRACT

Autonomic systems undergo dynamic compositional adaptation that often require *state transfer* and *synchronization* to correctly initialize the state of the new component, while ensuring that multiple fractions of the component are added and removed consistently. In general, if there are n different components for a given functionality, then there exist $n(n-1)$ possible adaptations for selecting an appropriate component. Identifying all these adaptations is not an easy task. Moreover, as verification of such adaptations is also difficult, it is desirable to reduce total number of these adaptations.

We propose a *component family* design for systematically building a repository of components from the perspective of dynamic adaptation. For a family of n components, we show that it suffices to identify n different adaptations. Moreover, to add a new component to this family, it suffices to consider only two adaptations. We also propose a design to separate the adaptation concern from component functionality for simplifying the specification and verification of adaptation. We introduce the *enhanced-primitive* relation between two components; when such a relation is known to exist, we show that it is possible to simplify the adaptation and its verification.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design and Tools—*modules and interfaces*; D.2.13 [**Software Engineering**]: Reusable Software—*reusable libraries*

General Terms

Design

Keywords

Dynamic Adaptation, Verification, Component-based Systems, Autonomic Computing

*This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, and a grant from Michigan State University.

DEAS 2005, May 21, 2005, St. Louis, Missouri, USA.

1. INTRODUCTION

Autonomic systems are often required to provide continuous and uninterrupted service. Especially in case of critical applications an interruption is highly undesirable and may very often be catastrophic. Moreover, with the changing requirements and/or execution environment, these systems need to adapt to the change. Further, to provide uninterrupted service, the adaptation should be done while the system continues to run.

Various techniques have been proposed in different research projects for adaptation. These approaches can be broadly classified (cf. [1]) based on the aspects of adaptation they address, such as (1) static *vs* dynamic (2) necessity of state transfer *vs* no state transfer (3) distributed systems *vs* single process systems (4) parameter adaptation *vs* compositional adaptation (5) application specific *vs* general purpose (6) anticipated *vs* unanticipated (7) language dependent *vs* language independent.

As component-based architecture is crucial for autonomic systems, we consider compositional adaptation that modifies the system dynamically (at run-time) by adding, removing, or replacing components. These components implement a part of the desired behavior of the system. For example, retransmission and forward error correction components are used to deal with message loss, or encryption components are used to provide security. These components are often distributed, i.e., the component consists of multiple *fractions* [2, 3], where each fraction is associated with one process in the distributed system. For example, a security component consists of a fraction that does encryption at the sender and a fraction that does decryption at the receiver. As a result, multiple processes are affected during adaptation.

While adding or removing such distributed components, all fractions of the component are not added or removed simultaneously. Hence, we will have a situation where some processes have added/removed the component fractions while some are yet to do that. In other words, fractions of different components may co-exist. Therefore, while adding or removing the fractions, the *dependency relationship* (cf. [2]) among the fractions should be handled correctly. In other words, it may be necessary to add/remove these component fractions in a certain order. For example, in case of a forward error correction component, if the encoder fraction at the sender is added before the decoder fraction is added at the receiver, then the messages might be lost or decoded incorrectly. Thus, *synchronization* among component fractions is required while adding or removing the component. Also, *state-transfer* between components may be required for the adaptation to be transparent to the application.

We say that an adaptation from component A to com-

ponent B exists if there is an approach that allows one to systematically replace all fractions of A by appropriate fractions of B. If such an approach has been verified so that the state transfer and synchronization issues are correctly handled then we say that a verified adaptation from A to B exists. One way to obtain a verified adaptation is to use the *transitional-invariant lattice* from [4]. In this approach, to replace component A by component B, we show that if we begin in a state where the initial program containing A starts in its invariant state then after the adaptation, the program containing B will also be in its invariant.

Now, consider the scenario where there are n different components that provide similar interface and functionality, and the choice of the component depends on application requirements and environment conditions. In this case, there are $n(n-1)$ possible adaptations among these components. Identifying the way in which all these adaptations can be performed and verifying them is difficult. Moreover, if the developer of the component wants to expose its details to only a subset of developers of other components then adaptation between that component and remaining components may not be possible. Thus, it is desirable to have a design methodology that helps in reducing the number of these adaptations, while still allowing adaptation between any two of these components.

Towards this end, we propose the notion of *component family* to limit the number of adaptations that need to be identified. This work is based on the ideas of family of programs as proposed by Parnas [5] and work on design of component-based software systems by Batory [6]. It focuses on building a family (library) of related components from the perspective of dynamic adaptation. The design of component family helps in: (1) simplifying the adaptation from one component to another, (2) facilitating the independent development of new components that can be used for adaptation, (3) enhancing the reusability of components, and (4) simplifying the verification of adaptation.

We also propose a design to separate adaptation concern from component functionality, i.e., to separate part of the component that provides the desired functionality and part of the component that is involved only during addition or removal of that component. This separation helps in automatically identifying and verifying adaptations, and enables modification of adaptation without affecting the component functionality. Our approach differs from previous work (e.g., [7]), where separation of application functionality and added components for adaptation is considered. While such separation is desirable in developing component-based systems, in this approach, the entire component is considered as an adaptive code. The separation proposed in our work is fine-grained, and is especially beneficial when one needs to specify and verify the necessary synchronization and state-transfer between components during adaptation. Specifically, such separation allows us to focus only on a subset of component implementation that is involved in adaptation.

We note that, in this paper, we do not address the issue of making an existing application *adapt-ready*. An application is defined to be adapt-ready, if it supports the adaptation by allowing addition, removal, or replacement of components. The common technique for making an application adapt-ready is often through some kind of indirection (e.g., [2, 7–11]). Further, we do not address "decision-making" , i.e., how an autonomic system selects an appropriate component. In this paper, we are interested in methodology that allows us to separate the adaptation concern, and simplifies the specification and verification of adaptation, while enabling independent development and reuse of components. Our approach for verification of adaptation is discussed in [4].

Organization of the paper. In Section 2, we discuss the structure and advantages of the component family. We discuss how the distributed program, component, and adaptation are modeled in Section 3. We present a case study discussing the example of a component family of reliable communication components in Section 4. In Section 5, we answer some of the questions related to our work on component family. Finally, we conclude in Section 6.

2. COMPONENT FAMILY

There are various components available for a given functional requirement. For example, to provide reliable communication one can use a proactive component based on forward error correction, or a reactive component based on message acknowledgments and retransmission. The key motivation for adaptation in autonomic systems is to provide an appropriate component for a given functionality based on the application requirements and environment conditions.

To perform adaptation from one component to another, there are two important issues, namely, state-transfer and synchronization, that need to be addressed. As discussed in Section 1, providing adaptation from one arbitrary component to another arbitrary component is difficult. Moreover, for the case where one does not know the details of some components, it is impossible to provide assurance for adaptations to and from those components. Further, while developing new component one does not know about all the existing components that can potentially be replaced by the new component. Also, it is difficult to anticipate the components that can be developed later. To overcome these limitations and to simplify the use of components in adaptation, we propose a design to build a *systematic* repository of components that provide similar functionality. Towards this end, we define the notion of a component family.

Definition. A component family is a strongly connected directed graph, say (V, E), where (i) each vertex in V denotes a component, (ii) all components have compatible interfaces and provide a similar functionality, and hence, any component in the family can be replaced by another, and (iii) Each arc $(v_1, v_2) \in E$ denotes that there exists an adaptation (that provides the necessary synchronization and state-transfer) from v_1 to v_2.

We illustrate a component family in Figure 1. Consider the subgraph consisting of vertices A, B, C, and D, which are the four components in the family. Direct adaptations are defined for each arc in this graph. The arc from A to D denotes that there exists an adaptation from A to D. We say that the arc (A, D) is a verified arc if verified adaptation from A to D exists. Further, as shown in the Figure 1, there exists a path from component A to component B, which implies that there exists a sequence of adaptations through which A can be replaced by B, namely, A to D, D to C, and C to B.

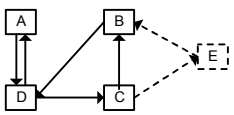

Figure 1: An Example of a Component Family

2.1 Component Structure

As discussed earlier in this section, in order to keep a graph of component family strongly connected, we need at least two adaptations associated with each component, such that one adaptation is to the component and one adaptation is from the component. For each adaptation, the component may perform some state-transfer and synchronization related actions, that are specific to that adaptation. In other words, a component has different state-transfer and synchronization related actions corresponding to different adaptations that it is involved in. However, the part of the component that performs the actual functionality remains the same irrespective of the adaptations.

Reckoning the adaptation requirements, each component in a component family is designed to consist of two parts: (i) a *functional* part, and (ii) an *adapt-active* part. The adapt-active part is involved in state-transfer and synchronization related actions that are needed only during adaptation. In other words, the functions of the adapt-active part are invoked only during adaptation.

Each component in a family consists of exactly one functional part. A component may have zero or more adapt-active parts. The adapt-active part of a component corresponds to a particular adaptation the component is involved in. A component may not have an adapt-active part (in other words, have an empty adapt-active part), if the component does not perform any state-transfer or synchronization related actions during adaptation. Also, a component may have an adapt-active part that is shared with multiple adaptations that it is associated with. From an implementation perspective, depending on the adaptation that the component is involved in, the appropriate adapt-active part corresponding to that adaptation should be loaded before adaptation. This can be triggered internally (by some monitoring module) or externally (by an user).

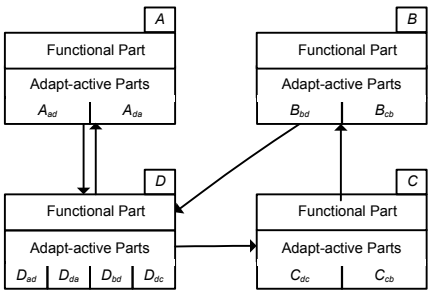

Figure 2: Structure of a Component in a Family

For example, consider the components A, B, C, and D in a component family as shown in Figure 2. The component B has two adapt-active parts, B_{bd} and B_{cb}. The part B_{bd} is used during adaptation from B to D, and the part B_{cb} is used during adaptation from C to B.

2.2 Advantages of Component Family

1. Simplifying adaptation between components and enabling independent development of new components. Consider the case where an application is using a component from a component family $\mathcal{F}$ consisting of n components. In this case, to provide adaptation between any two components of a family $\mathcal{F}$, we need a minimum of only n adaptations (in this case, the graph consists of a directed cycle). When a new component is developed that will be a part of $\mathcal{F}$, it suffices to have only two more adaptations while still keeping the graph strongly connected. For example, if a new component E is added to the component family shown in Figure 1, only two adaptations, say from C to E and from E to B, are enough to keep the graph strongly connected.

2. Simplifying verification of adaptation. To verify that the adaptation between components is correct (e.g., by using the approach in [4]), it is required that after adaptation the component continues to correctly perform its functionality, and specification during adaptation is satisfied. The separation of adapt-active parts from component functionality simplifies the task of specifying and verifying adaptation. Further, if the number of such adaptations is low, then less verification needs to be performed.

3. Reusability of components and adaptations. The design of component family not only enhances the reuse of components but also promotes the reuse of adaptations between components. For example, consider two components X and Y and that the adaptation from X to Y exists. Now, consider a component Z and the adaptation from Z to Y. In this case, by providing the adaptation from Z to X, the adaptation from Z to Y can be done in two steps while reusing the adaptation that already exists from X to Y. We note that if the direct adaptation from Z to Y were to exist, it would not necessarily be fast or simple. In fact, there are cases, as discussed in next point, where a two (or more) step adaptation is simpler than a direct adaptation between components.

4. Simplifying adaptation in case of an *enhanced-primitive* relationship among components. The state-transfer and synchronization during adaptation is in general difficult between arbitrary components. However, if one component is an enhancement of another component, then the issue of state-transfer and synchronization can be simplified in adaptation between those two components (cf. Section 4 for an example). To take advantage of this, we define the enhanced-primitive relationship between components. We say that a component A is an *enhanced* component of component B (respectively, component B is a *primitive* component of A) iff A is syntactically and semantically compatible with B, i.e., it extends the interface of B, and it provides all services that B provides.

Now, consider a scenario where A is an enhanced component of B and that B is being replaced by A. In this scenario, fractions of B can be replaced in an arbitrary order by fractions of component A, as each fraction of A can provide the required service to the remaining fractions of B. Moreover, the fractions of A can communicate with the remaining fractions of B using a protocol that the latter understands. Thus, in this case, the synchronization requirement among component fractions is relaxed, and also transferring state to/from primitive component is easy.

In a case where two enhanced components, say C and D, are derived from the same primitive component, say Z, the adaptation from C to D can be done in two steps; by first replacing C by Z and then replacing Z by D. Since the adaptation for enhanced-primitive relationship is relatively easy, the direct adaptation from C to D may not necessarily be fast or easy. We note that the idea of an enhanced-primitive relationship can be extended to have a multi-level hierarchy of components, where components at a higher level are enhanced version of components at lower level.

3. MODELING ADAPTATION

In this section, we describe how we model the distributed

programs, components, and adaptations. Though, our implementation of component family is in Java, for brevity, we choose an abstract model based on guarded commands for modeling. This modeling is based on the work in [12].

Program and Process. A program $\mathcal{P}$ is specified by a set of global constants, a set of global variables, and a set of processes. A process p is specified by a set of local constants, a set of local variables, and a finite set of *actions* (defined later in this section). Variables declared in $\mathcal{P}$ can be read and written by the actions of all processes. The processes in a program communicate with one another by sending and receiving messages.

Component and its fractions. A component is a set of global constants, a set of global variables, and a finite set of fractions that are involved in providing a common functionality. A component fraction is a set of local constants, a set of local variables, and a finite set of actions that are associated with a single process. A component and a fraction also has input parameters, which are values of the variables supplied by the program; and output parameters, which are values returned to the program.

Action. An action of process p is uniquely identified by a name, and is of the form

$$\langle name \rangle \ : \ \langle guard \rangle \ \rightarrow \ \langle statement \rangle$$

A guard is a combination of one or more of the following: a state predicate of p, a receiving guard of p, or a timeout guard. A state predicate of p is a boolean expression over the constants and variables of p. A receiving guard of p is of the form `rcv` $\langle message \rangle$ `from` $\langle q \rangle$. A timeout guard is of the form `timeout` $\langle$*state predicate of* $\mathcal{P}\rangle$. The statement of an action updates zero or more variables and/or sends one or more messages. An action can be executed only if its guard evaluates to true. To execute an action, the statement of that action is executed atomically. A sending statement of p is of the form `send` $\langle message \rangle$ `to` $\langle q_1, ..., q_n \rangle$, which sends a message to one or more processes . We say that an action of p is enabled in a state of $\mathcal{P}$ iff its guard evaluates to true in that state.

Atomic adaptation. As discussed in Section 1, to add a component to a program, each fraction of the component needs to be added at processes of the program. Similarly, to remove a component, each fraction of the component needs to be removed from the process of the program. In other words, adaptation in distributed programs involves multiple steps. We divide this multi-step adaptation into multiple *atomic adaptations*, such that each atomic adaptation involves only one process (fraction). In this work, we consider the following types of atomic adaptations: (i) `block` - block a fraction, (ii) `add` - add a fraction, (iii) `sadd` - add a fraction while initializing its state (to a previous state of the old component), (iv) `remove` - remove a fraction, and (v) `sremove` - remove a fraction while storing its state information. Thus, an adaptation consists of sequence of atomic adaptations. Each atomic adaptation is represented by a name and has a guard. An atomic adaptation is performed when the guard corresponding to it becomes true.

Remark on notation. When defining an atomic adaptation or an action, we use the **bold** font. And, when using them as a part of a predicate, we use sens serif (cf. Figures 4−7).

4. CASE STUDY

In this Section, we discuss the component family consisting of components that provide reliable communication. There are various components available for providing reliable communication. We consider three components in this family (cf. Figure 3), namely, (i) forward error correction (*fec*) component, (ii) acknowledgment (*ack*) component, and (iii) forward error correction with acknowledgment (*fecAck*) component. For simplicity, we discuss only one sender and one receiver; however, the case study can be easily extended for multiple sender and multiple receivers.

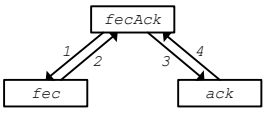

Figure 3: Reliable Communication Component Family

Forward Error Correction (*fec*) Component. The *fec* component is used to deal with message loss by sending extra packets. The receiver can recover the lost packets without requesting retransmission of lost packets. The *fec* component consists of two types of fractions: encoder and decoder. The encoder fraction is added at the sender process and the decoder fraction is added at the receiver process. The encoder takes $(n-k)$ data packets and encodes them to add k parity packets. It then sends the group of n (data and parity) packets. The decoder needs to receive at least $(n-k)$ packets of a group to decode all the data packets. Thus, if less than k packets in a group are lost then the receiver can recover the lost packets by using the extra packets from that group.

Component *fec*

Constants n, k, w

Fraction *s_fec*

Input parameters sQ, r
Variables g, p, u, eQ

encode : $eQ[u, 0..n-1] = \text{Empty} \wedge \texttt{count}(sQ) \geq n-k$
→ $eQ[u, 0..n-1] := \texttt{fec\_encode}(\texttt{head}(sQ, n-k))$;
$u := (u+1) \bmod w$;

send : $eQ[g, p] \neq \text{Empty}$
→ `send` $eQ[g, p]$ to r;
$eQ[g, p] := \text{Empty}$;
$p := (p+1) \bmod n$;
if $p = 0$
$g := (g+1) \bmod w$;

Fraction *r_fec*

Input parameters rQ
Variables $g_d, rbuf, wait, discard, undel\_grp$

receive : `rcv` data(g, p, x) from s
→ while ($wait$ = false);
if $discard(g)$ = false
$rbuf[g, p] := x$;
$undel\_grp[g] :=$ true;
if $discard[(g+1) \bmod w]$ = true
$discard[(g+1) \bmod w] :=$ false;

decode : $\texttt{count}(undel\_grp[0..w-1] \neq \text{Empty}) > 0$
→ if $\texttt{count}(rbuf[g_d, 0..n-1] \neq \text{Empty}) \geq n-k$
$wait :=$ true;
$rQ := rQ \circ \texttt{fec\_decode}(rbuf[g_d, 0..n-1])$;
if $\texttt{count}(rbuf[g_d, 0..n-1] \neq \text{Empty}) < n$
$discard[g_d] :=$ true;
$rbuf[g_d, 0..n-1] := \text{Empty}$;
$undel\_grp[g_d] :=$ false;
$g_d := (g_d+1) \bmod w$;
$wait :=$ false;

Figure 4: Forward Error Correction Component

The abstract version of the *fec* component is shown in Figure 4, which consists of the encoder fraction *s_fec* at the

Component *ack*

Constants n, w

Fraction *s_ack*

Input parameters sQ, r
Variables p, g, snt

send : $sQ \neq$ Empty $\wedge$ $snt[g, p] \neq$ Empty
→ $snt[g, p]$:= data(g, p, head(sQ));
send $snt[g, p]$ to r;
$p := (p + 1) \bmod n$;
if $p = 0$
$g := (g + 1) \bmod w$;

send_again : rcv nack(g_{na}, p_{na}) from r
→ if $snt[g_{na}, p_{na}] \neq$ Empty
send $snt[g_{na}, p_{na}]$ to r;

ack_rcv : rcv ack(g_a) from r
→ $snt[g_a, 0..n-1]$:= Empty;

Fraction *r_ack*

Input parameters rQ, s
Variables $i, g_d, rbuf, undel\_grp$

receive : rcv data(g, p, x) from s
→ $rbuf[g, p] := x$;
$undel\_grp[g]$:= true;

deliver : count($undel\_grp[0..w-1] \neq$ Empty) > 0
→ if count($rbuf[g_d, 0..n-1] \neq$ Empty) $= n$
$rQ := rQ \circ rbuf[g_d, 0..n-1]$;
$rbuf[g_d, 0..n-1]$:= Empty;
send ack(g_d) to s;
$undel\_grp[g_d]$:= false;
$g_d := (g_d + 1) \bmod w$;

send_nack : count($undel\_grp[0..w-1] \neq$ Empty) > 2
→ for $i = 0$ to $n-1$
if $rbuf[g_d, i]$ = Empty $\wedge$
$undel\_grp[g_d]$ = true
send nack(g_d, i) to s;

Figure 5: Acknowledgment Component

Component *fecAck*

Constants n, k, w
Variables ack_mode
Fraction *s_fecAck*

Input parameters sQ, r
Variables u, g, p, eQ, snt

encode : $eQ[u,0..n-1]$ = Empty $\wedge$ count(sQ) $\geq n - k$
→ $eQ[u,0..n-1]$:= fec_encode(head(sQ,$n-k$));
$u := (u + 1) \bmod w$;

send : $eQ[g, p] \neq$ Empty $\wedge$
($snt[g, p]$ = Empty $\vee$ $ack\_mode = false$)
→ $snt[g, p]$:= data(g, p, $eQ[g, p]$);
send $eQ[g, p]$ to r;
$eQ[g, p]$:= Empty;
$p := (p + 1) \bmod n$;
if $p = 0$
$g := (g + 1) \bmod w$;

send_again : rcv nack(g_{na}, p_{na}) from r
→ if $snt[g_{na}, p_{na}] \neq$ Empty
send $snt[g_{na}, p_{na}]$ to r;

ack_rcv : rcv ack(g_a) from r
→ $snt[g_a, 0..n-1]$:= Empty;

Fraction *r_fecAck*

Input parameters rQ, s
Variables $i, g_d, rbuf, wait, discard, undel\_grp$

receive : rcv data(g, p, x) from s
→ while ($wait$ = false);
if $discard(g)$ = false
$rbuf[g, p] := x$;
$undel\_grp[g]$:= true;
if $discard[(g + 1) \bmod w]$ = true
$discard[(g + 1) \bmod w]$:= false;

decode : count($undel\_grp[0..w-1] \neq$ Empty) > 0
→ if count($rbuf[g_d, 0..n-1] \neq$ Empty) $\geq n - k$
$wait$:= true;
$rQ := rQ \circ$ fec_decode($rbuf[g_d, 0..n-1]$);
if count($rbuf[g_d, 0..n-1] \neq$ Empty) $< n$
$discard[g_d]$:= true;
$rbuf[g_d, 0..n-1]$:= Empty;
if $ack\_mode = true$
send ack(g_d) to s;
$undel\_grp[g_d]$:= false;
$g_d := (g_d + 1) \bmod w$;
$wait$:= false;

send_nack : ack_mode = true $\wedge$
count($undel\_grp[0..w-1] \neq$ Empty) > 2
→ for $i = 0$ to $n-1$
if $rbuf[g_d, i]$ = Empty $\wedge undel\_grp[g_d]$ = true;
send nack(g_d, i) to s;

Figure 6: Forward Error Correction with Acknowledgment Component

sender process s and the decoder fraction ***r_fec*** at the receiver process r. The encoder fraction takes sQ and r as the input parameters; sQ being the sequence of messages that the sender process needs to send to the receiver process r. The decoder fraction takes rQ as the input parameter, which stores the messages received from the sender. The function `fec_encode` encodes the data packets, and the function `fec_decode` decodes the encoded data packets. The function head(sQ) returns the message at the front of sQ, and head(sQ, m) returns m messages from the front of sQ. The `count` function is used to count the number of entries in the list (array). The notation $sQ \circ d$ denotes the concatenation of sQ and $\langle d \rangle$.

Acknowledgment (*ack*) Component. The ***ack*** component deals with message loss by retransmitting the lost packets. It uses acknowledgment to confirm the receipt of message sent by the sender, and negative acknowledgment to confirm the loss of message sent by the sender. It consists of an encoder fraction at the sender and a decoder fraction at the receiver. The encoder fraction adds a group and a packet number in each packet. It maintains a window of size w and sends all packets in that window to the receiver. It waits for acknowledgment of receipt of a group before moving the window one group forward. If it receives a negative acknowledgment for any packet, it sends that packet again to the decoder. When the decoder receives a packet out of order, it waits for few more packets before sending a negative acknowledgment to the encoder fraction. When all packets in a group are received, it sends an acknowledgment for that group to the encoder.

The abstract version of the ***ack*** component is shown in Figure 5, which consists of the encoder fraction ***s_ack*** at the sender process s and the decoder fraction ***r_ack*** at the receiver process r. Similar to the ***fec*** component, the encoder has sQ and r as the input parameters. The decoder has rQ and s as the input parameters.

Forward Error Correction with Acknowledgment (*fecAck*) Component. The ***fecAck*** component uses both forward error correction and acknowledgments. If the rate of message loss is high and more than k packets are lost in a group, then negative acknowledgments can be used for retransmission of the lost packets. If the rate of message loss is low and less than k packets are lost in a group, then the receiver can recover the lost packets without requesting any retransmission.

The abstract version of the ***fecAck*** component is shown in Figure 6, which consists of the fraction ***s_fecAck*** at the sender process and the fraction ***r_fecAck*** at the receiver process.

Adaptations. We now consider the adaptations that exists in this family as shown in Figure 3. There are four adap-

Adapt-active parts for adaptation 1 : *fecAck* to *fec*
Fraction : *s_fecAck*
a_{11s} : true → *ack_mode* := false
a_{12s} : a_{11s} ∧ a_{11r} → sremove *s_fecAck* $[n, k, w, u, g, p, eQ]$
Fraction : *r_fecAck*
a_{11r} : true → *ack_mode* := false
a_{12r} : a_{11s} ∧ a_{11r} → sremove *r_fecAck* $[n, k, w, g_d, rbuf, undel\_grp, discard, wait]$
Fraction : *s_fec*
a_{12s} : a_{12s}(*s_fecAck*) → sadd *s_fec* $[n, k, w, u, g, p, eQ]$
Fraction : *r_fec*
a_{12r} : a_{12r}(*r_fecAck*) → sadd *r_fec* $[n, k, w, g_d, rbuf, undel\_grp, discard, wait]$

Adapt-active parts for adaptation 2 : *fec* to *fecAck*
Fraction : *s_fec*
a_{21s} : true → sremove *s_fec* $[n, k, w, u, g, p, eQ]$
Fraction : *r_fec*
a_{21r} : true → sremove *r_fec* $[n, k, w, g_d, rbuf, undel\_grp, discard, wait]$
Fraction : *s_fecAck*
a_{21s} : a_{21s}(*s_fec*) → sadd *s_fecAck* $[n, k, w, u, g, p, eQ]$
Fraction : *r_fecAck*
a_{21r} : a_{21r}(*r_fec*) → sadd *r_fecAck* $[n, k, w, g_d, rbuf, undel\_grp, discard, wait]$
a_{22r} : a_{21s} → *ack_mode* := true

Adapt-active parts for adaptation 3 : *fecAck* to *ack*
Fraction : *s_fecAck*
a_{31s} : true → block encode
a_{33s} : a_{32r} → remove *s_fecAck*
Fraction : *r_fecAck*
a_{32r} : a_{31s} ∧ *snt* = Empty ∧ *undel_grp* = Empty → remove *r_fecAck*
Fraction : *s_ack*
a_{34s} : a_{33s} ∧ a_{33r} → add *s_ack*
Fraction : *r_ack*
a_{33r} : a_{32r} → add *r_ack*

Adapt-active parts for adaptation 4 : *ack* to *fecAck*
Fraction : *s_ack*
a_{41s} : true → block send
a_{43s} : a_{42r} → remove *s_ack*
Fraction : *r_ack*
a_{42r} : a_{41s} ∧ *snt* = Empty ∧ *undel_grp* = Empty → remove *r_ack*
Fraction : *s_fecAck*
a_{44s} : a_{43s} ∧ a_{43r} → add *s_fecAck*
Fraction : *r_fecAck*
a_{43r} : a_{42r} → add *r_fecAck*

Figure 7: Adaptations in Reliable Communication Component Family

tations that exist in this family, namely, (1) *fecAck* to *fec*, (2) *fec* to *fecAck*, (3) *fecAck* to *ack*, and (4) *ack* to *fecAck*. The arcs in the graph are labeled accordingly. (Note that maximum possible adaptations in a family of three components is six, and minimum required adaptations to keep the graph strongly connected is three).

The adapt-active parts of each fractions that are involved in adaptation are shown in Figure 7. The adapt-active parts of the fractions contain the atomic adaptations that are associated with them. The name of each atomic adaptation has three subscripts (e.g., a_{13s}). The first subscript denotes the adaptation (cf. label of the arc in Figure 3). The second subscript denotes the order in the sequence of atomic adaptations. If two atomic adaptations have the same second subscript, it means that they can be executed in any order. The third subscript denotes the process that the atomic adaptation is associated with. For example, the atomic adaptation a_{13s} denotes that in the adaptation 1 (*fecAck* to *fec*), it is the third in the sequence of atomic adaptations and it occurs at process s.

Adaptation 1*: fecAck to fec having* **enhanced-primitive relationship**. Consider the *fec* and *fecAck* components as shown in Figures 4 and 6. The *fecAck* component is syntactically compatible with *fec* and it also provides all the services that *fec* provides. In this case, *fecAck* is an enhanced component of *fec*, which is a primitive component. During adaptation that replaces *fecAck* to *fec*, the fractions can be changed arbitrarily, provided (*i*) *fecAck* component is running in a primitive mode before the adaptation begins, and (*ii*) state of *fecAck* component is transferred to *fec* component. As shown in Figure 7, first the *ack_mode* is set to false (a_{11s} and a_{11r}); as a result the *fecAck* component is now running in primitive mode, i.e., in a mode compatible with *fec*. Now, the fractions can be changed arbitrarily (a_{12s} and a_{12r}). There are two atomic adaptations named a_{12s}, one associated with fraction *s_fecAck* and another associated with fraction *s_fec*. This implies that sremove and sadd needs to be done atomically, i.e., removal of *s_fecAck* and addition of *s_fec* needs to be done in an atomic manner. Similarly, removal of *r_fec* and addition of *r_fecAck* needs to be done in an atomic manner. Note that sremove and sadd carry an extra argument, which represents the state information that is transferred from the existing component to the new component. In this case, the state of *fecAck* component is transferred to the *fec* component.

Adaptation 2*: fec to fecAck having* **primitive-enhanced relationship**. As discussed in the previous adaptation, in this case, the fractions can be changed arbitrarily, as the components share a primitive-enhanced relationship. We need to ensure that (*i*) state of *fec* component is transferred to *fecAck* component, and (*ii*) *fecAck* runs in a primitive mode till the adaptation is complete. Initially, the *ack_mode* is set to false, so *fecAck* runs in a mode that is compatible with *fec*. After the adaptation is complete, *ack_mode* is set to true. The adaptation from *fec* to *fecAck* is as shown in Figure 7.

Adaptation 3*: fecAck to ack*. The adaptation that replaces *fecAck* to *ack* is as shown in Figure 7. Unlike adaptations 1 and 2, the components *fecAck* and *ack* do not share an enhanced-primitive relationship, because the *fecAck* component is not designed to run in *ack*-only mode. Specifically, the send function in *fecAck* is not compatible with that of *ack*. Hence, *fecAck* component does not provide all the functionality that *ack* component does. Therefore, the fractions cannot be changed arbitrarily. First the encoder fraction *s_fecAck* at the sender s needs to be blocked from encoding more packets (a_{31s}). Once all the packets that are already encoded are sent by s, and received by the receiver r, then decoder fraction *r_fecAck* can be removed (a_{32r}). After the *r_fecAck* fraction is removed, the removal of the *s_fecAck* fraction (a_{33s}) and the addition of the *r_ack* fraction (a_{33r}) can be done in any order. Finally, after the *s_fecAck* fraction at s is removed and *r_ack* fraction at r is added, the encoder fraction *s_ack* is added at s (a_{34s}). (Note that *fecAck* and *ack* could be modified so that they satisfy the enhanced-primitive relationship. We have chosen not to do so to illustrate the case where the components are not related by an enhanced-primitive relationship.)

Adaptation 4*: ack to fecAck*. The adaptation that replaces *ack* to *fecAck* is similar to the adaptation 3 discussed above. Since the components do not share an enhanced-primitive relationship, the fractions need to be changed in a specific order as shown in Figure 7.

5. DISCUSSION

In this section, we answer some of the questions related to our work on component family.

Can there be multiple adaptation paths between two compo-

nents? If yes, then which adaptation path should be chosen?

Yes. There can be multiple paths between two components. In this case, additional factors could be taken into account while deciding the appropriate path for adaptation. To this end, for each arc, we could associate several factors, e.g., the time or resources required for adaptation, types of faults that could be tolerated in the adaptation. Based on the factors associated with each arc, we can compute the characteristics for different paths. These characteristics can then be used to determine the suitable path.

How do we develop components that satisfy the enhanced-primitive relationship?

One way to design such components is to use inheritance. Inheritance provides syntactic compatibility between components. To ensure semantic compatibility, we need to extend the inheritance relationship, such that, a derived component provides all services that a parent component provides.

How does the component family design help when adaptation involves components providing different functionalities, say security and reliability?

In this case, there will be two separate component families, namely, a family of components that provide reliability and a family of components that provide security. The application will have to perform separate adaptations for reliability and security components.

There may exist a scenario, although undesirable as it violates the principle of separation of concerns, where some component, say C, provides reliability as well as security. In this scenario, C will be present in both the families. Now, if the application that is using C to provide security and reliability decides to use only a security component, then the application can perform adaptation to replace C with a security component. Also, existence of such components could be used in adaptations where one needs to trade off between two desirable properties such as reliability and security. Specifically, if the application were to replace a reliability component by a security component (may be because of environment changes that require it to have security but where reliability cannot be provided due to other constraints such as energy management) then the application could first replace the reliability component with C and then replace C by the security component.

How can we perform the adaptation where some component is removed although not replaced by other component?

If such a scenario is desired for a particular component family then that family should have a *default component* which is equivalent to having no component at all. For example, in the context of our case study in Section 4, the default component would be one that provides no recovery for lost messages. Thus, removal of a component is equivalent to replacing that component by the default component. This approach is similar to that in [2].

6. CONCLUSION

In this paper, we presented a design methodology for building a systematic repository of components from the perspective of dynamic adaptation. The component family architecture helps in reducing the number of adaptations among components of a family. The design also helps to provide adaptation between components, even if details of these components are not available to each other. In this case, the adaptation between these two components is done through a sequence of adaptations that involve other components.

Moreover, the component family design helps in simplifying the verification task, by separating the adaptation concern from component functionality. Also, the reduced number of adaptations reduces the number of verifications that need to be done. Furthermore, the adaptation and verification is made simpler in case where the components share an enhanced-primitive relationship.

We have implemented the component family of reliable communication components in Java. Using the reset-based framework that we developed in [2], we perform adaptations between components of this family. In [3], we have also discussed an example of tree correction components, which share an enhanced-primitive relationship.

The separation of adaptation concern from component functionality is important towards realizing the goal of making the component-based systems autonomously and dynamically adapt to their changing environment and requirements, and to verify such adaptations. In Section 4, we presented an example of reliable communication components family and showed how adaptation can be specified separately from component functionality. As a part of future work, we plan to automatically generate adaptation code from adaptation specification.

As discussed in Section 5, in a component family, multiple adaptations may exist between two components. In this case, the choice for the adaptation depends on various factors such as time, speed, resource availability, and fault-tolerance requirements. As a part of future work, we plan to develop a heuristic to assign weights to arcs in the graph of the family based on these factors. The weights assigned to adaptations (arcs) can be used to find an optimal adaptation between two components.

7. REFERENCES

[1] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Gunter Kniesel. Towards a taxonomy of software change. *Software Maintenance and Evolution: Research and Practice*, 2003.

[2] Sandeep S. Kulkarni, Karun N. Biyani, and Umamaheswaran Arumugam. Composing distributed fault-tolerance components. In *Workshop on Principles of Dependable Systems, DSN*, June 2003.

[3] Karun Biyani. Dynamic composition of distributed components. Master's thesis, Michigan State University, 2003.

[4] Sandeep Kulkarni and Karun Biyani. Correctness of component-based adaptation. In *International Symposium on Component-based Software Engineering*, May 2004.

[5] Daniel M. Hoffman and David M. Weiss, editors. *Software Fundamentals - Collected Papers by David L. Parnas.* Addison-Wesley, 2001.

[6] Don Batory, Roberto Lopez Herrejon, and Jean-Phillipe Martin. Generating product-lines of product families. In *Automated Software Engineering Conference*, 2002.

[7] S. Masoud Sadjadi. *Transparent Shaping of Existing Software to Support Pervasive and Autonomic Computing.* PhD thesis, Michigan State University, 2004.

[8] W. K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *21st International Conference on Distributed Computing Systems*, pages 635–643, April 2001.

[9] J. Hallstrom, W. Leal, and A. Arora. Scalable evolution of highly available systems. *Transactions of the Institute for Electronics, Information and Communication Engineers*, To appear.

[10] P. McKinley and U. Padmanabhan. Design of composable proxy filters for mobile computing. In *Workshop on Wireless Networks and Mobile Computing*, 2001.

[11] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application. In *ECOOP*, 2002.

[12] M. Gouda. *Elements of Network Protocol Desgin.* John Wiley & Sons, 1998.