Assurance of Dynamic Adaptation in Distributed Systems ⋆

Karun N. Biyani Sandeep S. Kulkarni

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824 USA

Abstract

Long running applications often need to adapt due to changing requirements or changing environment. Typically, such adaptation is performed by dynamically adding or removing components. In these type of adaptations, components are often added to or removed from multiple processes in the system. As a result, during adaptation, the system may consist of both changed and unchanged processes, causing *old* and *new* components to overlap. This overlapping of components during adaptation may induce cross-component communication, which may lead to behavior during adaptation that is unpredictable and/or undesirable.

In this paper, we discuss an approach to model and verify *overlap adaptation*. We use *transitional-invariant lattice* and *transitional-faultspan lattice* to verify correctness of adaptation in absence and presence of faults, respectively. We also discuss framework to support implementation of overlap adaptation.

Key words: Dynamic Adaptation, Assurance, Correctness, Specification, Verification, Fault-Tolerance

1 Introduction

Software systems need to adapt as the requirements and/or execution environment change. Stopping the system during adaptation is often undesirable, as it may be in-

⋆ The preliminary version of this paper was published in [1].

Email: {biyanika,sandeep}@cse.msu.edu, Tel: +1-517-355-2387

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.

convenient and/or potentially unsafe to interrupt the running system. In other words, adaptation needs to be performed while the system continues to operate. Such adaptation is commonly referred to as *dynamic adaptation*.

Adaptive software provides techniques (e.g. [2–9]) that allow the software to modify its own functional behavior or non-functional behavior (e.g., its fault-tolerance, quality of service or security requirements). These modifications may include reconfiguration of some parameters, or addition or removal of application code. A survey in [10] presents various tools and techniques in building adaptive software. In component-based systems, adaptation is often performed by adding, removing or replacing components of the system.

To gain assurance about the adaptation, formal specification and verification of adaptation is crucial. In context of dynamic adaptive systems, there are three aspects of verification: (*i*) verifying system before adaptation, (*ii*) verifying system during adaptation, and (*iii*) verifying system after adaptation. While existing verification techniques can be used to verify system before and system after adaptation, such techniques cannot be applied directly to verify the system during adaptation. This is because during adaptation the system (and possibly its specification) is changing whereas existing work assumes that the system and its specification remain unchanged.

In case of distributed systems, multiple processes need to be changed during adaptation. In such cases, changes to multiple processes need to be synchronized and interactions between changed and unchanged processes need to be controlled. We call adaptation in distributed systems as *overlap adaptation* when behavior of *old program* (program before adaptation) and *new program* (program after adaptation) overlaps during adaptation. We classify overlap adaptation into three main categories: (*i*) *mixed-mode* adaptation, (*ii*) *quiescence* adaptation, and (*iii*) *parallel* adaptation. In case of quiescence adaptation, which is the most common approach for adaptation in distributed systems, there is no interaction allowed between the old and the new component during adaptation. During adaptation the old and the new components may exists in the system simultaneously, but individual processes are changed such that processes using old component fractions do not communicate with processes using new component fractions. In contrast, in case of mixed-mode adaptation, the old component and the new component are allowed to interact. In case of parallel adaptation, each node has both the old and the new component, but communication across components is not allowed; old (respectively, new) component fraction at a process can communicate with old (respectively, new) component fractions at other processes. In case of *non-overlap* adaptation in distributed systems, first the old component is removed from all processes before the new component is added, thereby, preventing any overlap of behavior of the old component

and the new component during adaptation.

We show in [11] that mixed-mode adaptation is better in terms of performance. Specifically, the time taken to complete adaptation is less compared to quiescence form of adaptation, thereby, reducing the service interruption time during adaptation. Moreover, the number of messages that need to exchanged among processes for synchronization is lesser in mixed-mode adaptation compared to quiescence adaptation. This is specifically important in systems, such as wireless and sensor networks, where message communication is costly. In spite of these advantages mixed-mode adaptation has not been addressed adequately in the literature due to challenges involved in verification and implementation.

With the above motivation, in this paper, we present an approach to formally specify and verify overlap adaptation. Our approach can be applied to both quiescence and mixed-mode adaptation. The approach discussed in this paper can also be used to specify and verify non-overlap adaptation. Since adaptation is often used to add (extend) fault-tolerance properties to a given program, we also focus on methods for verification of fault-tolerance properties during adaptation. Hence, we extend our approach to verify fault-tolerance properties during adaptation.

Numerous techniques have been proposed to address various issues in formalizing adaptation. A survey in [12] discuss various approaches based on graphs, process algebras, logic and other formalisms used to specify adaptive systems. Most of the approaches [2, 3, 5, 6, 8, 13–19] focus on formalizing the structural design and implementation of adaptive systems. Graph-based approaches [13–15] use graph rewriting rules to specify dynamism. Approaches in [16, 17] use variety of process algebras such as Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP), and π-calculus. Architectural Description Language (ADL) based approaches [18–20] model programs as components and connectors, and adaptation as reconfiguration of connections. Compared to these approaches, our approach is based on simple guarded commands and focus on verifying the behavior of system during adaptation.

Other approaches that have addressed the issue of verifying adaptation include [21–24]. The approaches in [21–23] focus on offline adaptation, whereas approach in [24] focuses on online adaptation of a single process system (that can also be extended to distributed systems that communicate via RPC). However, none of these approaches explicitly focus on the behavior of system during (overlap) adaptation in distributed systems.

Contributions of the paper. The main contributions of the paper are as follows:

- We introduce *adaptation lattice* to specify the behavior of system during adaptation.

- We introduce *transitional-invariant lattice* to verify correctness of adaptation in the absence of faults.
- We extend the transitional-invariant lattice to *transitional-faultspan lattice* to verify correctness of adaptation in the presence of faults.
- To illustrate our approach, we consider the addition of a forward error correction based *proactive component* (cf. Sect. 4), and the replacement of the proactive component by an acknowledgment based *reactive component* (cf. Sect. 6) in context of a message communication application.
- We briefly describe the extension to the distributed reset-based framework [4] to support implementation of correct adaptation.

Organization of the paper. The rest of the paper is organized as follows: In Sect. 2, we introduce formal definitions to model an adaptive system, adaptation, and define specification during adaptation. In Sect. 3, we introduce the notion of transitional-invariant lattice to verify adaptation in the absence of faults, and illustrate its use in Sect. 4 by verifying the addition of the proactive component to the message communication application. In Sect. 5, we introduce the notion of transitional-faultspan lattice to verify adaptation in the presence of faults and in Sect. 6, we illustrate its use by verifying replacement of the proactive component with the reactive component in the presence of faults. In Sect. 7, we present the architecture framework to support the implementation of adaptation. We discuss some of the issues concerning our work in Sect. 8, and finally, conclude in Sect. 9.

2 Modeling Adaptation

In this section, we introduce a formal model for adaptation in asynchronous programs. We consider program as an automaton. Informally, adaptation of a program can be described as transforming the automaton to another automaton. We first discuss the informal overview of adaptation in distributed systems and then present the formal model to reason about the correctness of adaptation. In the rest of the paper, for sake of brevity, we use the term adaptation to mean *overlap* adaptation. However, our approach also applies to non-overlap adaptation.

We refer to the program before adaptation as the *old program* and program after adaptation as the *new program*. An adaptation replaces the old program being executed by the system with the new program. We assume that the old program and the new program are independently correct, i.e., by themselves they can execute and produce acceptable behavior. The goal of verifying adaptation is to ensure that: (i) the adaptation ends in a state from where the system satisfies the behavior of the new program,

and (*ii*) the behavior during adaptation is acceptable (as defined by specification during adaptation).

An adaptation in a distributed system involves multiple steps that are executed at various processes. We consider the replacement of a fraction at a single process as an atomic step of adaptation. The key to verifying adaptation is to ensure that the atomic steps in adaptation occur in an appropriate order and the instances when they occur are "safe", i.e., the specification during adaptation is satisfied. We denote each atomic step of adaptation as *atomic adaptation* (defined formally later).

To verify adaptation, the ordering among atomic adaptations and the behavior during adaptation needs to be verified. To verify the behavior during adaptation we need to classify the states of the program during adaptation. The intermediate states that occur during adaptation are due to overlapping of the old program and the new program. The properties satisfied by these intermediate states may be different from either the old program or the new program. Consequently, the behavior expected during adaptation needs to be specified separately from the old program and the new program. Towards this end, we define the notion of *intermediate program*. The intermediate program arise due to overlapping of behavior of the old program and the new program. The first atomic adaptation modifies the old program into the first intermediate program. Similarly, other atomic adaptations modifies one intermediate program into the next intermediate program. The last atomic adaptation results into the new program. The specification during adaptation identifies the requirements for these intermediate programs.

2.1 Abstract Model of Adaptation

We model a program as an automaton $\mathcal{A}$ represented as a tuple $\langle S, \Sigma, \delta, S_0 \rangle$, where

- $S(\mathcal{A})$ - a set of states
- $\Sigma(\mathcal{A})$ - a set of actions
- $\delta(\mathcal{A})$ - a state-transition relation, where $\delta(\mathcal{A}) \subseteq S(\mathcal{A}) \times \Sigma(\mathcal{A}) \times S(\mathcal{A})$
- $S_0(\mathcal{A})$ - a nonempty subset of $S(\mathcal{A})$ known as initial states

Each element (s, π, s') of $\delta(\mathcal{A})$ is known as a *transition*, where $s, s' \in S(\mathcal{A})$ and $\pi \in \Sigma(\mathcal{A})$. If $\mathcal{A}$ has a transition (s, π, s') it means that π is *enabled* in state s and executing action π in state s will lead to state s'. A transition of the form $(s,_{,} s')$ denotes that $\exists \pi : \pi \in \Sigma(\mathcal{A}) : (s, \pi, s') \in \delta(\mathcal{A})$.

We model an adaptation Δ using a 5-tuple as follows:

- $\mathcal{I}$ - a set of automata
- P - an automaton of the old program, $P \in \mathcal{I}$
- Q - an automaton of the new program, $Q \in \mathcal{I}$
- Σ_a - a set of special type of actions known as *adaptive actions*
- S_{map} - a state mapping is a partial function $S(\mathcal{A}) \times \Sigma_a \rightarrow S(\mathcal{A}')$ and $\mathcal{A}, \mathcal{A}' \in \mathcal{I}, \mathcal{A} \neq \mathcal{A}'$

The old program, the new program, and all intermediate programs are modeled as automata. Given an adaptive action, the state mapping defines the states of the automaton in which the adaptive action can execute, and corresponding states of the resulting automaton in which the adaptive action terminates. Note that the state mapping is a partial function, as it may not be possible to perform corresponding atomic adaptation in all states of the automaton. Each element $((s, \pi_a), s')$ of S_{map} can be represented as a triplet (s, π_a, s'). Similar to the state-transition relation of an automaton, a state mapping S_{map} can be defined as a subset of $S(\mathcal{A}) \times \Sigma_a \times S(\mathcal{A}')$ with the restriction that if $(s, \pi_a, s') \in S_{map}$ and $(s, \pi_a, s'') \in S_{map}$ then $s' = s''$. Each element of S_{map} is known as an *adaptive transition*.

Broadly speaking, the state mapping of adaptation Δ defines an *automata-transformation* (partial) function $\delta_a : \mathcal{I} \times \Sigma_a \rightarrow \mathcal{I}$. Each element $((\mathcal{A}, \pi_a), \mathcal{A}')$ (equivalently, $(\mathcal{A}, \pi_a, \mathcal{A}')$) of δ_a is known as *atomic adaptation*. Thus, each atomic adaptation is modeled as transforming one automaton to another automaton.

The automata-transformation relation represents an adaptation lattice defined as follows:

Adaptation Lattice. An *adaptation lattice* (cf. Fig. 1) is a finite directed acyclic graph in which each node is labeled with an automaton and each edge is labeled with an atomic adaptation, such that,

(1) There is a single *start node* P having no incoming edges. The start node is associated with the automaton representing the old program. The automata-transformation function (correspondingly, S_{map}) satisfies the following condition:
$\forall \mathcal{A}, \pi_a :: (\mathcal{A}, \pi_a, P) \notin \delta_a$

(2) There is a single *end node* Q having no outgoing edges. The end node is associated with the automaton representing the new program. The automata-transformation function (correspondingly, S_{map}) satisfies the following condition:
$\forall \mathcal{A}, \pi_a :: (Q, \pi_a, \mathcal{A}) \notin \delta_a$

(3) Each intermediate node R has at least one incoming edge and at least one outgoing edge. It is associated with the automaton representing the intermediate program. The automata-transformation (correspondingly, S_{map}) satisfies the following condition:

$$\forall \mathcal{A} : \mathcal{A} \neq P : (\exists \mathcal{A}', \pi_a :: (\mathcal{A}', \pi_a, \mathcal{A}) \in \delta_a) \wedge$$
$$\forall \mathcal{A} : \mathcal{A} \neq Q : (\exists \mathcal{A}', \pi_a :: (\mathcal{A}, \pi_a, \mathcal{A}') \in \delta_a)$$

A path in the lattice from the start node to the end node is called an *adaptation path*.

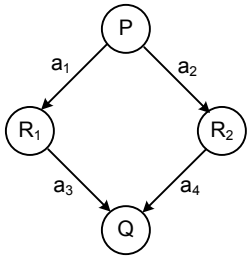


Fig. 1. An example of a adaptation lattice.

Based on the above definition, an adaptation can also be viewed as an automaton, where

- $S(\Delta) = \bigcup_{\mathcal{A} \in \mathcal{I}} \{(\mathcal{A}, s) \mid s \in S(\mathcal{A})\}$
- $\Sigma(\Delta) = \bigcup_{\mathcal{A} \in \mathcal{I}} \{(\mathcal{A}, \pi) \mid \pi \in \Sigma(\mathcal{A})\} \cup \Sigma_a$
- $\delta(\Delta) = \bigcup_{\mathcal{A} \in \mathcal{I}} \{((\mathcal{A}, s), (\mathcal{A}, \pi), (\mathcal{A}, s')) \mid (s, \pi, s') \in \delta(\mathcal{A})\} \cup$
  $\{((\mathcal{A}, s), \pi_a, (\mathcal{A}', s') \mid (s, \pi_a, s') \in S_{map}\}$
- $S_0(\Delta) \subseteq S(\mathcal{P})$

We now introduce some notations and terminology used in specifying and verifying adaptive programs.

Enables. An action π of $\mathcal{A}$ is *enabled* in state s if $\exists s' :: (s, \pi, s') \in \delta(\mathcal{A})$. An adaptive action π_a is enabled in $\mathcal{A}$ if $\exists s \in \mathcal{A}, s' \in \mathcal{A}' :: (s, \pi_a, s') \in S_{map}(\Delta)$.

State predicate. A state predicate X of $\mathcal{A}$ is any subset of $S(\mathcal{A})$. We say a X is true in state s if $s \in X$.

Closure. A state predicate X of $\mathcal{A}$ is closed in $\mathcal{A}$ (respectively, $\delta(\mathcal{A}), \Sigma(\mathcal{A})$) iff the following condition holds:

$$\forall s, s', \pi :: ((s, \pi, s') \in \delta(\mathcal{A})) \Rightarrow (s \in X \Rightarrow s' \in X)$$

Computation. A computation of program $\mathcal{A}$ (respectively, adaptation Δ) is a sequence of states $\sigma = \langle s_0, s_1, ... \rangle$ satisfying the following conditions:

- For first state s_0 in σ, $s_0 \in S_0(\mathcal{A})$ (respectively, $S_0(\Delta)$)
- If σ is infinite then $\forall j : j > 0 : (\exists \pi :: (s_{j-1}, \pi, s_j) \in \delta(\mathcal{A}))$ (respectively, $\delta(\Delta)$)

- If σ is finite and terminates in state s_l, then there does not exist state s for all π such that $(s_l, \pi, s) \in \delta(\mathcal{A})$ (respectively, $\delta(\Delta)$), and $\forall j : j > 0 : (\exists \pi :: (s_{j-1}, \pi, s_j) \in \delta(\mathcal{A}))$ (respectively, $\delta(\Delta)$)

Specification. A specification of $\mathcal{A}$ is a set of *acceptable* computations. Following Alpern and Schneider [25], specification can be decomposed into a safety specification and a liveness specification. As shown in [26], for a rich class of specifications, safety specification can be represented as a set of bad transitions that must not occur in program computations; i.e., safety specification is a subset of $S(\mathcal{A}) \times S(\mathcal{A})$.

Satisfies. $\mathcal{A}$ satisfies specification if each computation of $\mathcal{A}$ is in specification. $\mathcal{A}$ satisfies specification from X iff (i) X is closed in $\mathcal{A}$, and (ii) each computation of $\mathcal{A}$ is in specification and starts from a state where X is true (i.e., $S_0(A) \subseteq X$).

Invariant. The state predicate X of $\mathcal{A}$ is an invariant iff $\mathcal{A}$ satisfies specification from X. Note that $X \supseteq S_0(\mathcal{A})$. Informally speaking, the invariant predicate characterizes the set of all states reached in the "correct" computations of $\mathcal{A}$.

Safety specification during adaptation. Similar to the specification of $\mathcal{A}$, safety specification during adaptation Δ is specified as a set of bad transitions that must not occur in computations of adaptation Δ, i.e., as a subset of $S(\Delta) \times S(\Delta)$.

Liveness specification during adaptation. We argue that the specification during adaptation should be a safety specification. This is due to the fact that one often wants the adaptation to be completed as quickly as possible. Hence, it is desirable not to delay the adaptation task to satisfy the liveness specification during adaptation. Rather, it is desirable to guarantee that, after adaptation, the program reaches states from where its (new) safety and liveness specification is satisfied. Thus, the implicit liveness specification during adaptation is that the adaptation completes. In other words, the liveness specification during adaptation is that each intermediate program eventually executes its adaptive action. For this reasons, we have omitted the representation of liveness specification of program.

3 Verifying Adaptation in Absence of Faults

In this section, we introduce the notion of *transitional-invariant lattice* to verify the correctness of adaptation. The idea of transitional-invariant lattice is based on the concept of *proof lattice* [27], which is used to prove liveness properties of a concurrent program.

As discussed in Section 2, the program during adaptation consists of actions of the old program and actions of the new program. Therefore, we consider intermediate programs obtained after one or more atomic adaptations. Similar to the invariants that are used to identify "legal" program states and are closed under program execution, we define *transitional-invariants*.

Transitional-invariant. Let R be an intermediate program in the adaptation Δ. A *transitional-invariant* is a predicate that is closed in R.

Note that the actions of intermediate program are the old program actions that are not yet removed and the new program actions that are already added. However, the adaptive actions do not necessarily preserve the transitional-invariant. Now, we define *transitional-invariant lattice*.

Transitional-invariant lattice. A *transitional-invariant lattice* is an adaptation lattice with each node having one predicate and that satisfies the following five conditions:

1. **Safety of old program**. The start node P is associated with an invariant S_P of the program before adaptation.
2. **Safety of new program**. The end node Q is associated with an invariant S_Q of the program after adaptation.
3. **Safety of intermediate program**. Each intermediate node R is associated with a predicate TS_R that is a transitional-invariant for any intermediate program at R (i.e., intermediate program obtained by performing adaptations from the entry node to R). Furthermore, any intermediate program at R satisfies the (safety) specification during adaptation from TS_R.
4. **Safety of adaptive action**. If a node labeled R_i has an outgoing edge labeled a to a node labeled R_j, then for all adaptive transitions (s, a, s') in S_{map} where TS_{R_i} is true in state s, TS_{R_j} is true in state s'. Furthermore, all the adaptive transitions (s, a, s') satisfies the safety specification during adaptation.
5. **Progress of adaptation**. If a node labeled R has outgoing edges labeled $a_1, a_2, ..., a_k$ to nodes labeled $R_1, R_2, ..., R_k$, respectively, then in all computations of adaptation there exists a transition (s, s') such that for some $i : 1 \leq i \leq k : (s, a_i, s') \in S_{map}$. Furthermore, $\forall s : s \in TS_R : (\forall a, s' : a \in \Sigma_a - \{a_1, ..., a_k\} : (s, a, s') \notin S_{map})$.

Correctness of Adaptation. Intuitively, an adaptation is correct if the following conditions are satisfied: If the adaptation begins in a legitimate state of the old program then during adaptation safety during adaptation is met and the resulting state of the new program is legitimate. With this intuition, if adaptation begins in a state where invariant of the old program is true, then we say that adaptation is correct if:

- Adaptation terminates in a state where invariant of the new program is true
- During adaptation safety specification during adaptation is satisfied
- Eventually adaptation terminates

The following theorem states that finding a transitional-invariant lattice is necessary and sufficient for proving correctness of adaptation.

Theorem 1. *Given S_P as the invariant of the program before adaptation and S_Q as the invariant of the program after adaptation, the adaptation from P to Q is correct if and only if there is a transitional-invariant lattice for the adaptation with start node associated with S_P and end node associated with S_Q.*

Proof.

($\Rightarrow$) If transitional-invariant lattice exists then adaptation is correct.

If the stated conditions are satisfied, then the specification of old program is satisfied when the adaptation starts. Also, the existence of transitional-invariant lattice during adaptation ensures that for each intermediate program that occurs during adaptation, the specification during adaptation is satisfied. Moreover, from the definition of transitional-invariant lattice, each adaptive action satisfies safety specification during adaptation. Also, in each intermediate program eventually some adaptive action will be executed, which ensures liveness of adaptation. Furthermore, the last adaptive action terminates in the invariant of the new program, from where the system satisfies the behavior of the new program. Thus, the existence of transitional-invariant lattice proves correctness of adaptation.

($\Leftarrow$) If adaptation is correct then transitional-invariant lattice exists (proof by construction).

Assuming the adaptation is correct, to show the existence of the lattice, we proceed as follows. First, we identify the structure of the lattice. Then, for each node, we identify the corresponding transitional-invariant.

By definition, the transitional invariant lattice has one entry node P that is associated with S_P, the invariant of the old program and one exit node associated with S_Q, the invariant of the new program. For the following discussion, let the entry node be denoted as current node.

Now, consider all computations C of adaptation that start from the transitional-invariant associated with the current node. For each of these computations, identify the computation prefix until first occurrence of the atomic adaptation (including the state/program

reached after the atomic adaptation). This occurrence exists since correctness of adaptation implies that eventually the program would be changed to the new program. Let C_P denote the set of these computation prefixes. If a_i is an atomic adaptation occurring in this set of computation prefixes then add an edge from the current node to a new node, say R_i. The label on the edge from the current node to R_i would be a_i.

To identify the invariant associated with R_i, proceed as follows: The initial value of the invariant is TR_i-*init*, where TR_i-*init* denotes the set of all states in C that occur after the atomic adaptation a_i. Now, consider all computations of the program at R_i (i.e., these computations do not include the atomic adaptation) that start from TR_i-*init*. The set of states reached in this set of computation identifies the transitional-invariant TR_i at node R_i. Now, this process is repeated for all possible atomic adaptations in C_P; this will identify the new nodes and corresponding programs and transitional-invariants at those nodes.

The above process is repeated for all newly created nodes as well. If the atomic adaptation being considered is the last adaptation in the multi-step adaptation process (i.e., the resulting program would be the new program) then the successor node is Q.

By construction, we can see that the transitional-invariant at each node is closed and the computation of the intermediate program from that transitional-invariant will result in execution of one of the *permitted* atomic adaptation and the resulting state would satisfy the constraint of the lattice. Moreover, since the adaptation is correct, when the last atomic adaptation is performed, the resulting state must be in S_Q. Thus, the constructed lattice meets all the constraints of the transitional-invariant lattice. □

4 Example: Message Communication

In this section, we present an example that illustrates how transitional-invariant lattice can be used to verify correctness of adaptation in the context of a message communication program. We first discuss the programming notation used to describe the system. We then describe the fault-intolerant message communication program and the FEC-based *proactive component*. Next, we discuss adaptation of adding a proactive component to the fault-intolerant message communication program. We then discuss (in Sect. 6) adaptation of replacing the proactive component with an acknowledgment-based *reactive component*.

4.1 Programming Notation

In this subsection, we discuss the programming notation we use to describe the system. For brevity, we express programs using guarded commands [28–30]. This gives a compact representation of the program defined in Sect. 2 (in terms of state space and transitions). It is straightforward to translate from the compact representation of the program to its automata representation discussed in Sect. 2 as we discuss in this subsection.

Program and process. A program $\mathcal{P}$ is specified by a finite set of processes and channels. A process p is specified by a set of variables and a finite set of *actions*. The processes in a program communicate with one another by sending and receiving messages over unbounded channels that connect the processes. A channel from process p to process q is denoted by a channel variable $C_{p,q}$, which is an unbounded queue. Only process p can append an item of data to the rear of the queue $C_{p,q}$ and only process q can delete an item at the head of the queue $C_{p,q}$. Each variable has a predefined nonempty domain. A state of a process is obtained by assigning each variable a value from its respective domain. The state of the channel connecting p and q is given by the value of the queue $C_{p,q}$. The state of the program is given by the state of all the processes and the channels. Thus, the state space of the program $\mathcal{P}$, $S(\mathcal{P})$, is the set of all possible states of $\mathcal{P}$. We use $s(x)$ to denote value of variable x in state s, and $V(p)$ to denote the set of variables of process p. The state predicate of $\mathcal{P}$ is a boolean expression over process and channel variables.

Note that a state predicate may be characterized by the set of all states in which its boolean expression is true and, therefore, is a subset of the state space of the program.

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

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

A guard of each action is a boolean expression over the process and the channel variables. The statement of each action is such that its execution updates zero or more process or channel variables. The set of actions of the program $\mathcal{P}$, $\Sigma(\mathcal{P})$ is given by the set of names of all the actions of all the processes of $\mathcal{P}$. Each action of p gives a set of transitions of the form (s, π, s') such that the guard of action π is true in state s and execution of statement of π in s results in state s'. Thus, the state-transition relation $\delta(\mathcal{P})$ is obtained from the set of actions of all the processes of $\mathcal{P}$.

Component and fraction. A component is specified by a finite set of fractions that are involved in providing a common functionality. Intuitively, a component implements a part of the desired behavior of the system, such as some algorithm or protocol. A

component fraction is specified by a set of variables and a finite set of actions that are associated with a single process. A component (respectively, fraction) is syntactically same as a program (respectively, process), with only difference that some variables of the component are designated as *input*, whose values are supplied by the program with which it is composed. The composition of the component and the program is the union of the variables and actions of the component and the program.

Adaptive action. An adaptive action is a special type of action, which is identified by a unique name and is of the form

$$\langle name \rangle \ : \ \langle guard \rangle \ \rightarrow \ TransformTo(p', \Phi).$$

When the statement of the adaptive action is executed, the current process is replaced by p' and state-mapping Φ is used to initialize the variables of p'. Each adaptive action π_a gives a set of adaptive transitions of the form (s, π_a, s') such that the guard of π_a is true in state s of process p and execution of the statement of π_a results in state $s' = \Phi(s)$ of process p'. The state mapping function $S_{map}(\Delta)$ is obtained from the set of all adaptive actions.

From modeling perspective, we consider that the adaptive action replaces the entire process, even if only a small part of it is actually changed. In actual implementation, the adaptive action can performed in various ways, such as blocking of some method, or loading/unloading of some class. However, from verification point of view it is only important to consider the effect of the adaptive action. Additionally, considering each adaptive action as a generic form of process replacement gives the developer freedom to implement the adaptive action based on the platform and the language used.

State mapping. We define the following classes of state mapping, Φ, that occur during atomic adaptation:

- *Identity mapping.* In identity mapping, the names and values of the variables remain the same. Formally, for a variable y of $V(p')$, there exists a variable y of $V(p)$ such that for all s, $(\Phi(s))(y) = s(y)$.
- *Quasi mapping.* In quasi mapping, the name of the variable of the new process is different from that of the old process, though its value is same as the value of some equivalent variable in the old process state when the adaptive action is executed. Formally, for a variable y of $V(p')$, there exists a variable x of $V(p)$ such that for all s, $(\Phi(s))(y) = s(x)$.
- *Initial mapping.* In initial mapping, the variables of the new process are initialized to some value as in the initial state of the new process. Formally, for a variable y of $V(p')$, for all s, $(\Phi(s))(y) = y_0$, where $y_0 \in S_0(y)$ and $S_0(y)$ is the set of values from

domain of y that y can take in the initial states of process p'.

- *Functional Mapping.* In functional mapping, the value of the variable of the new process is some function of the values of variables of the old process. Formally, for a variable y of $V(p')$, for all s, $(\Phi(s))(y) = f(V(p))$.
- *Arbitrary Mapping.* A special type of functional mapping is arbitrary mapping, where all variables of the new process are assigned some arbitrary value. Formally, for a variable y of $V(p')$, for all s, $(\Phi(s))(y) = y_d$, where $y_d \in D(y)$ and $D(y)$ denotes the domain of variable y.
- *Mixed Mapping.* Most mapping that occur in practice are mixed mapping, in which variables of the new process $V(p')$ are divided into disjoint sets, and one of the above mappings is associated with each set.

Notation. We use "." to denote the *belongs to* relation. For example, if variable v belongs to process p, it is denoted by $p.v$, and action a of process p is denoted by $p.a$. A process p of program $\mathcal{P}$ is denoted by $\mathcal{P}.p$, and a fraction i of component C is denoted by $C.i$. For brevity, we avoid using belongs to relation if it is obvious from the context.

4.2 Fault-Intolerant Communication Program

Specification of the communication program. An infinite queue of messages at a sender process s is to be sent to two receiver processes r_1 and r_2 via two unicast channels and copied in corresponding infinite queues at the receivers. Faults may cause loss of messages in the channel.

program $\mathcal{P}_{intol}$
process s
  var sQ : **queue** of **integer**
      m : **integer**
  begin
    send : $\neg$isEmpty(sQ) $\rightarrow$ $m := \texttt{head}(sQ)$;
                $C_{s,r_1}, C_{s,r_2} := C_{s,r_1} \circ m, C_{s,r_2} \circ m$
  end
process $r_i[i = 1, 2]$
  var rQ : **queue** of **integer**
      m : **integer**
  begin
    receive : $\neg$isEmpty(C_{s,r_i}) $\rightarrow$ $m := \texttt{head}(C_{s,r_i})$;
                $rQ := rQ \circ m$
  end

Fig. 2. Message communication program (fault-intolerant version).

The message communication program is shown in Fig. 2. Only `send` and `receive` actions

of the program are shown, since only those actions are considered for adaptations.

Processes s, r_1, and r_2 maintain queues sQ, $r_1.rQ$, and $r_2.rQ$ respectively. sQ contains messages that s needs to send to r_1 and r_2. The messages received by r_i from s are stored in $r_i.rQ$. Let mQ be the queue of all messages to be sent. (mQ is an auxiliary variable that is used only for the proof.) Initially, $sQ = mQ$. The function $\texttt{head}(sQ)$ returns the message at the front of sQ, and $\texttt{head}(sQ, k)$ returns k messages from the front of sQ. The function $\texttt{type}(C_{s,r})$ returns the type of message at the head of channel queue. The notation $sQ \circ d$ denotes the concatenation of sQ and $\langle d \rangle$.

Invariant. The invariant of the communication program is $S_P = S1 \wedge S2$, where
$S1 = \forall i : (m_i \in r_1.rQ \vee m_i \in r_2.rQ) \Rightarrow m_i \in mQ$, and
$S2 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \veebar ((m_i \in C_{s,r_1} \veebar m_i \in r_1.rQ) \wedge (m_i \in C_{s,r_2} \veebar m_i \in r_2.rQ)))$.

In the above invariant, $S1$ indicates that messages received by the receivers are sent by the sender. $S2$ indicates that a message m_i is not yet sent by the sender, or it is in the channel, or it is already received by the receiver, all exclusive.

Notation. The symbol $\veebar$ denotes *exactly one* operator, i.e. $x \veebar y \veebar z$ implies exactly one of x, y and z is true.

4.3 Proactive Component

The proactive component sends extra messages to the receiver, which the receiver can use to recover from lost messages. It 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. This component provides tolerance to certain message loss faults (discussed in Sect. 6).

Fig. 3 shows the abstract version of the proactive component. The encoder and decoder fractions of the component are shown. The encoder fraction consists of two actions: `encode` and `fec_send`. The decoder consists of two actions: `decode` and `fec_receive`. These fractions are composed with the process that will use them. The composition of fraction and process is done by union, which is equivalent to appending the actions of the fraction and the process. We assume that appropriate renaming is performed so that there are no inconsistencies in definitions of the variables of the fractions and the processes. The message communication program composed with the proactive component

Component *fec*

Fraction *encoder*

inp sQ : **queue** of **integer**
r_1, r_2

var n, k, u, l, m : **integer** {initially, $u = l = m = 0$}
$encQ$: **array** [**integer**, $0..n-1$] of **integer** {initially, $encQ = \perp$}

begin

encode : $true \rightarrow encQ[u, 0..n-1] := \text{fec\_encode}(\texttt{head}(sQ, n-k));$
$u := u + 1$

[] **fec_send** : $encQ[l, m] \neq \perp \ \rightarrow \ C_{s,r_1}, C_{s,r_2} := C_{s,r_1} \circ encQ[l, m], C_{s,r_2} \circ encQ[l, m];$
$m := (m + 1)$ **mod** n;
if $m = 0$ **then**
$l := l + 1$
fi

end

Fraction $decoder_i$

inp rQ : **queue** of **integer**
s

var n, k, x, y, p, m : **integer** {initially, $p = 0$}
$rbufQ$: **array** [**integer**, $0..n-1$] of **integer** {initially, $rbufQ = \perp$}

begin

fec_receive : $\neg\text{isEmpty}(C_{s,r_i}) \ \rightarrow \ x, y, m := \texttt{head}(C_{s,r_i});$
$rbufQ[x, y] := m$

[] **decode** : $\texttt{count}(rbufQ[p, 0..n-1] = \perp) >= (n-k) \ \rightarrow$
$rQ := rQ \circ \text{fec\_decode}(rbufQ[p, 0..n-1]);$
$p := p + 1$

end

Fig. 3. Proactive component.

is shown in Fig. 4.

Program $\mathcal{P}_{fec}$

process s

var : $\mathcal{P}_{intol}.s.\text{var} \ \cup \ fec.encoder.\text{var}$

begin

fec.encoder.encode

[] **fec.encoder.fec_send**

end

process $r_i[i = 1, 2]$

var : $\mathcal{P}_{intol}.r_i.\text{var} \ \cup \ fec.decoder_i.\text{var}$

begin

fec.decoder.fec_receive

[] **fec.decoder.decode**

end

Fig. 4. Message communication program (with proactive component).

Specification of program using the proactive component. Program using the proactive component satisfies the same specification as the communication program (cf. Sect. 4.2).

Invariant. The invariant of the program using the proactive component is $S_Q = S1 \wedge S_F$, where

$S_F = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \veebar ((m_i \in r_1.rQ \veebar m_i \in \text{data}(encQ \cup C_{s,r_1} \cup r_1.rbufQ)) \wedge (m_i \in r_2.rQ \ \veebar \ m_i \in \text{data}(encQ \cup C_{s,r_2} \cup r_2.rbufQ))))$

We use the notation $m_i \in \text{data}(\text{encQ} \cup C_{s,r_1} \cup r_1.rbufQ)$ to imply that message m_i can be generated from the data in $\{\text{encQ} \cup C_{s,r_1} \cup r_1.rbufQ\}$. In the above invariant, S_F indicates that the message is either at the sender, or already received by the receiver, or it can be generated from the data in the channel and the buffers at the sender and the receiver.

4.4 Adaptation: Addition of the Proactive Component

Given a program shown in Fig. 2, the adaptation of adding the proactive component converts the program to one shown in Fig. 4. We first need to have adapt-ready version of the program $\mathcal{P}_{intol}$ as shown in Fig. 5.

program $\mathcal{P}_{a\text{-}intol}$
process s
  var : $\mathcal{P}_{intol}.s.$var
  begin
    $\mathcal{P}_{intol}.s.$send
  [] $a_1 : true \ \rightarrow \ TransformTo(\mathcal{P}_{a\text{-}ip_1}.s, \Phi_{a_1})$;
  end
process $r_i[i = 1, 2]$
  var rQ : **queue** of **integer**
  begin
    $\mathcal{P}_{intol}.r_i.$receive
  [] $a_{(i+1)} : a_1 \ \wedge \ \text{isEmpty}(C_{s,r_i}) \ \rightarrow \ transformTo(\mathcal{P}_{fec}.r_i, \Phi_{a_{(i+1)}})$;
  end

Fig. 5. Message communication program (fault-intolerant version, adapt-ready).

Specification during adaptation. The specification during adaptation is that S_1 continues to be true during adaptation.

program $\mathcal{P}_{a\text{-}ip_1}$
process s
  var : $\mathcal{P}_{intol}.s.$var
  begin
    $a_4 : a_2 \ \wedge \ a_3 \ \rightarrow \ TransformTo(\mathcal{P}_{fec}.s, \Phi_{a_4})$;
  end
process $r_i[i = 1, 2]$: same as in Fig. 5

Fig. 6. Intermediate program $\mathcal{P}_{a\text{-}ip_1}$.

We identify the intermediate programs during adaptation after each atomic adaptation. The execution of adaptive action a_1 in $\mathcal{P}_{a\text{-}intol}$ results in the intermediate program $\mathcal{P}_{a\text{-}ip_1}$ shown in Fig. 6. $\mathcal{P}_{a\text{-}ip_1}$ does not send any packets, but the packets that are there in the channel can still be received by the receivers r_1 and r_2. In the execution of $\mathcal{P}_{a\text{-}ip_1}$ eventually all the packets in the channel are read and no new packets are added in the

channel from sender to receiver. Thus, the guards of the adaptive actions a_2 and a_3 eventually get enabled. The transitional-invariant of $\mathcal{P}_{a\text{-}ip_1}$ is: $TS_1 = S_1 \wedge S_2$, where S_1, S_2 are as defined earlier in Section 4.2.

program $\mathcal{P}_{a\text{-}ip_2}$

process s : same as in Fig. 6

process r_1 : same as in Fig. 4

process r_2 : same as in Fig. 5

Fig. 7. Intermediate program $\mathcal{P}_{a\text{-}ip_2}$.

Since a_1 and a_2 occur independently, we consider both possible orderings among them. The execution of adaptive action a_2 in $\mathcal{P}_{a\text{-}ip_1}$ results in the intermediate program $\mathcal{P}_{a\text{-}ip_2}$ shown in Fig. 7. In $\mathcal{P}_{a\text{-}ip_2}$, receiver r_1 has replaced its fraction, whereas receiver r_2 has not yet replaced its fraction and can receive any remaining packets in the channel from s to r_2. Eventually, in the execution of $\mathcal{P}_{a\text{-}ip_2}$ the guard of adaptive action a_3 gets enabled and a_3 is executed resulting in the intermediate program $\mathcal{P}_{a\text{-}ip_4}$.

The transitional-invariant of $\mathcal{P}_{a\text{-}ip_2}$ is $TS_2 = S_1 \wedge S_3$, where
$S_3 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \underline{\vee} ((m_i \in r_1.rQ) \wedge (m_i \in C_{s,r_2} \underline{\vee} m_i \in r_2.rQ)) \wedge \text{isEmpty}(C_{s,r_1}) = true \wedge \text{isEmpty}(r_1.rbufQ) = true \wedge r_1.p = 0)$.

program $\mathcal{P}_{a\text{-}ip_3}$

process s : same as in Fig. 6

process r_1 : same as in Fig. 5

process r_2 : same as in Fig. 4

Fig. 8. Intermediate program $\mathcal{P}_{a\text{-}ip_3}$.

The execution of adaptive action a_3 in $\mathcal{P}_{a\text{-}ip_1}$ results in the intermediate program $\mathcal{P}_{a\text{-}ip_3}$ shown in Fig. 8. In $\mathcal{P}_{a\text{-}ip_3}$, receiver r_2 has replaced its fraction, whereas receiver r_1 has not yet replaced its fraction and can receive any remaining packets in the channel from s to r_1. Eventually, in the execution of $\mathcal{P}_{a\text{-}ip_3}$ the guard of adaptive action a_2 gets enabled and a_2 is executed resulting in intermediate program $\mathcal{P}_{a\text{-}ip_4}$.

The transitional-invariant of $\mathcal{P}_{a\text{-}ip_3}$ is $TS_3 = S_1 \wedge S_4$, where
$S_4 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \underline{\vee} ((m_i \in r_2.rQ) \wedge (m_i \in C_{s,r_1} \underline{\vee} m_i \in r_1.rQ)) \wedge \text{isEmpty}(C_{s,r_2}) = true \wedge \text{isEmpty}(r_2.rbufQ) = true \wedge r_2.p = 0)$.

program $\mathcal{P}_{a\text{-}ip_4}$

process s : same as in Fig. 6

process $r_i[i = 1, 2]$: same as in Fig. 4

Fig. 9. Intermediate program $\mathcal{P}_{a\text{-}ip_4}$.

In the intermediate program $\mathcal{P}_{a\text{-}ip_4}$ shown in Fig. 9, only the adaptive action a_4 is enabled, and execution of a_4 results in the new program $\mathcal{P}_{fec}$. The transitional-invariant of $\mathcal{P}_{a\text{-}ip_4}$ is $TS_4 = S_1 \wedge S_5$, where

$S_5 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \ \veebar \ (m_i \in r_1.rQ \ \wedge \ m_i \in r_2.rQ)) \ \wedge \ \text{isEmpty}(C_{s,r_1}) = true \wedge \text{isEmpty}(r_1.rbufQ) = true \wedge r_1.p = 0 \wedge \text{isEmpty}(C_{s,r_2}) = true \wedge \text{isEmpty}(r_2.rbufQ) = true \ \wedge \ r_2.p = 0.$

State mapping. The state mapping for each adaptive action is shown in Table 1. Each adaptive action initializes the state of the new process when it is executed based on this mapping.

Mapping Function	**Process Affected**	**New State**
Φ_{a_1}	s	Identity mapping
Φ_{a_2}	r_1	$\{rQ, s\}$ - Identity mapping, $\{n, k, x, y, p, m, rbufQ\}$ - Initial mapping
Φ_{a_3}	r_2	$\{rQ, s\}$ - Identity mapping, $\{n, k, x, y, p, m, rbufQ\}$ - Initial mapping
Φ_{a_4}	s	$\{sQ, r_1, r_2\}$ - Identity mapping, $\{n, k, u, l, m, encQ\}$ - Initial mapping

Table 1. State mapping for each adaptive action.

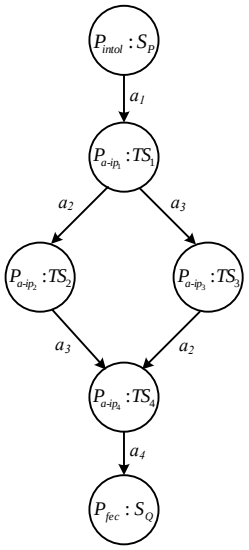


Fig. 10. Adaptation lattice for addition of proactive component.

Fig. 10 shows the adaptation lattice for the adaptation of adding the proactive component.

Theorem 1.1. *The adaptation lattice of Fig. 10 is a transitional-invariant lattice for the adaptation of adding the proactive component. Hence, the adaptation is correct.* □

5 Verifying Adaptation in Presence of Faults

In Sect. 3, we defined transitional-invariant lattice that is used to verify correctness of adaptation in absence of faults. In this section, we define *transitional-faultspan lattice* to verify correctness of adaptation in presence of faults. We first introduce some terms that we use use in this section. These definitions are based on the previous work in [29, 31]. Next, we define *transitional-faultspan* and *transitional-faultspan lattice*. Using these definitions, we present an approach to prove correctness of adaptation in presence of faults.

Fault class. Let Σ_f be a set of fault actions. A fault class $F(\mathcal{A})$ for program $\mathcal{A}$ is a subset of the set $S(\mathcal{A}) \times \Sigma_f \times S(\mathcal{A})$. We use $\mathcal{A}[]F$ to denote the transitions obtained by taking the union of the transitions in $\delta(\mathcal{A})$ and the transitions in $F(\mathcal{A})$. A fault class $F(\Delta)$ for adaptation Δ is:
$\bigcup_{\mathcal{A} \in \mathcal{I}} \{((\mathcal{A}, s), \Sigma_f, (\mathcal{A}, s')) \mid (s, \Sigma_f, s') \in F(\mathcal{A})\}$.

Fault-span. A state predicate T is a fault-span (F-span) of $\mathcal{A}$ from invariant S iff (i) $S \subseteq T$, and (ii) T is closed in $\mathcal{A}[]F$. Fault-span of a program identifies the set of states that a program can reach in presence of faults and asserts that the set of states is closed under the execution of program and fault actions.

Computation in presence of faults. A computation of program $\mathcal{A}$ (respectively, adaptation Δ) in presence of faults is a sequence of states $\sigma = \langle s_0, s_1, ... \rangle$ satisfying following conditions:

- For first state s_0 in σ, $s_0 \in S_0(\mathcal{P})$ (respectively, $S_0(\Delta)$)
- If σ is infinite then $\forall j : j > 0 : (\exists \pi :: (s_{j-1}, \pi, s_j) \in \delta(\mathcal{A}) \cup F(\mathcal{A}))$ (respectively, $\delta(\Delta) \cup F(\Delta)$)
- If σ is finite and terminates in state s_l, then there does not exist state s for all π such that $(s_l, \pi, s) \in \delta(\mathcal{A})$ (respectively, $\delta(\Delta)$), and $\forall j : j > 0 : (\exists \pi :: (s_{j-1}, \pi, s_j) \in \delta(\mathcal{A}))$ (respectively, $\delta(\Delta)$)
- $\exists n : n \geq 0 : (\forall j : j > n : (\exists \pi :: (s_{j-1}, \pi, s_j) \in \delta(\mathcal{A}))$ (respectively, $\delta(\Delta)$)

The second requirement captures that in each step, either a program (respectively, program or adaptive) transition or a fault transition is executed. The third requirement captures that faults do not have to execute. Finally, the fourth requirement captures that the number of fault-occurrences in a computation is finite. This requirement is the same as that made in previous work [32–35] to ensure that eventually recovery can occur.

Fault-tolerance (F-tolerant). $\mathcal{A}$ is F-tolerant for specification *spec* from S iff the following two conditions hold: $(i)\mathcal{A}$ satisfies *spec* from S, and (ii) there exists T such that T is an F-span of $\mathcal{A}$ from S, and every computation of $\mathcal{A}[]F$ starting in a state where T is true satisfies safety specification.

Remark. Henceforth, whenever the invariant S and the program $\mathcal{A}$ are clear from the context, we will omit them; thus, "T is a F-span of $\mathcal{A}$ from S for *spec*" abbreviates to "T is a F-span".

Let F_P be the fault class of the old program (i.e., the program before adaptation) and F_Q be the fault class of the new program (i.e., the program after adaptation). Let S_P be an invariant and T_P be a F_P-span of the old program. Similarly, let S_Q be an invariant and T_Q be a F_Q-span of the new program. The old program is F_P-tolerant, and the new program is F_Q-tolerant. Let F be the fault class during adaptation.

In the context of adaptation, we define *transitional-faultspans* to identify the set of states that the program can reach while performing adaptation in presence of faults.

Transitional-faultspan. Let R be an intermediate program in the adaptation Δ, and TS be a transitional-invariant of R. A *transitional-faultspan* (F-span) of R from TS is a predicate TT that satisfies following two conditions: $(i)\, TS \subseteq TT$, and $(ii)\, TT$ is closed in $R[]F$.

Now, we define *transitional-faultspan lattice*.

Transitional-faultspan lattice. A *transitional-faultspan (F-span) lattice* is an adaptation lattice where each node is associated with two predicates, a transitional-invariant and a transitional-faultspan, and the following conditions are satisfied:

0. **Correctness in absence of faults**. The adaptation lattice obtained by considering the transitional-invariants only forms a transitional-invariant lattice.
1. **Fault-tolerance of old program**. The entry node P is associated with a F_P-span T_P of the program before adaptation.
2. **Fault-tolerance of new program**. The exit node Q is associated with a F_Q-span T_Q of the program after adaptation.
3. **Fault-tolerance of intermediate program**. Each intermediate node R is associated with a predicate TT_R that is a transitional-faultspan (F-span) from TS_R for any intermediate program at R (i.e., intermediate program obtained by performing adaptations from the entry node to R). Furthermore, any intermediate program at R is F-tolerant from TS_R.
4. **Safety of adaptive action**. If a node labeled R_i has an outgoing edge labeled a

to a node labeled R_j, then for all adaptive transitions (s, a, s') in S_{map} where TT_{R_i} is true in state s, TT_{R_j} is true in state s'. Furthermore, all the adaptive transitions (s, a, s') satisfies the safety specification during adaptation.

5. **Progress of adaptation**. If a node labeled R has outgoing edges labeled $a_1, a_2, ..., a_k$ to nodes $R_1, R_2, ..., R_k$, respectively, then in all computations of adaptation there exists a transition (s, s') such that for some $i : 1 \leq i \leq k : (s, a_i, s') \in S_{map}$. Furthermore, $\forall s : s \in TT_R : (\forall a, s' : a \in \Sigma_a - \{a_1, ..., a_k\} : (s, a, s') \notin S_{map})$.

Correctness of adaptation in presence of faults. Intuitively, an adaptation is correct in presence of faults F if the following conditions are satisfied: If the adaptation begins in a legitimate state of the old program then during adaptation each intermediate program is F-tolerant and the resulting state of the new program is legitimate. With this intuition, if adaptation begins in a state where fault-span of the old program is true, then we say that adaptation is correct if:

- Adaptation terminates in a state where fault-span of the new program is true
- During adaptation, each intermediate program is F-tolerant
- Eventually adaptation terminates

The following theorem states that finding a transitional-faultspan lattice is necessary and sufficient for proving correctness of adaptation.

Theorem 2. *Given S_P as the invariant of the program before adaptation, T_P as the faultspan used to show that the program before adaptation is tolerant to F_P, S_Q as the invariant of the program after adaptation, and T_Q as the faultspan used to show that the program after adaptation is tolerant to F_Q, the adaptation from P to Q is correct in presence of faults F if and only if there is a transitional-faultspan (F-span) lattice for the adaptation with start node associated with S_P and T_P, and end node associated with S_Q and T_Q.*

Proof. The proof of this theorem is similar to the proof of Theorem 1 discussed in Section 3. □

Remark. Different types of tolerance specifications that normally occur in practice, namely, *masking*, *fail-safe*, and *non-masking* tolerance have been considered in the previous work [26,29,31]. In above discussion, we have considered masking fault-tolerance. The definitions can be easily extended to consider the fail-safe and non-masking tolerance during adaptation.

5.1 Adaptation of Self-Stabilizing Programs

In this section, we consider the adaptation considered in [36] where the authors have focused on adapting from one self-stabilizing program into another self-stabilizing program [32]. We show that this is an instance of our approach where all the transitional-faultspan predicates are *true*.

A program is self-stabilizing if starting from an arbitrary state it eventually recovers to a legitimate state. Thus, in transforming from one stabilizing program to another, we can let all the fault-spans (i.e., fault-span of the old program, fault-span of the new program and transitional-faultspans of intermediate programs) be true. With this approach, if the old program starts in any state, eventually the new program execution begins although the state of the new program may be arbitrary. Since the new program is self-stabilizing, it will eventually recover to legitimate states.

Note that in [36] the corresponding transitional-invariants may not exist. Specifically, even if the old program begins in legitimate states, the new program may initially be in illegitimate states before recovery takes place. Moreover, [36] allows arbitrary behavior during adaptation and, hence, the specification during adaptation may not be met.

6 Example: Message Communication (Continued)

In this section, we continue with the example of Sect. 4. We consider the adaptation that replaces the proactive component with the reactive component. We first discuss the adapt-ready version of the proactive component and the faults tolerated by the proactive component. Next, we describe the acknowledgment-based reactive component. We then discuss the adaptation of replacing the proactive component by the reactive component. Finally, we identify the transitional-faultspan lattice to verify correctness of this adaptation in presence of faults.

6.1 Proactive Component

We discussed the proactive component in Sect. 4.3. The adapt-ready version of the proactive component is shown in Fig. 11.

The specification of the program using proactive component is discussed in Sect. 4.3. Additionally, it tolerates message loss faults of class $F1$ (cf. Fig. 12). Faults of class $F1$ causes a loss of up to k messages in a group. In writing the fault transitions, we use auxiliary variables m^g to denote a message m from group g, and $lostCount_i^g$ to

Program $\mathcal{P}_{aa\text{-}fec}$
process s
  var : $\mathcal{P}_{fec}.s.\text{var}$
  begin
      fec.encoder.encode
  [] **fec.encoder.fec_send**
  [] $aa_1 : true \ \rightarrow \ transformTo(\mathcal{P}_{aa\text{-}ip_1}.s, \Phi_{aa_1})$
  end
process $r_i[i = 1, 2]$
  var : $\mathcal{P}_{fec}.r_i.\text{var}$
  begin
      fec.decoder.fec_receive
  [] **fec.decoder.decode**
  [] $aa_{(i+2)} : aa_2 \wedge \text{isEmpty}(C_{s,r_i}) \ \rightarrow \ transformTo(\mathcal{P}_{ack}.r_i, \Phi_{aa_{(i+2)}})$
  end

Fig. 11. Message communication program (with proactive component, adapt-ready).

denote the number of messages lost in group g in the channel from s to r_i. Initially, $\forall g :: lostCount_i^g = 0$.

$\textbf{msg\_loss}_i : m^g \in C_{s,r_i} \ \wedge lostCount_i^g \leq k \ \rightarrow \ C_{s,r_i} := C_{s,r_i} - m^g;$
$lostCount_i^g + +$

Fig. 12. Fault class $F1$

Fault-span. The $F1$-span of the program using the proactive component is $T_Q = S_Q$. The fault-span is same as the invariant since the proactive component provides masking fault-tolerance.

6.2 Reactive Component

The reactive component deals with message loss by retransmitting the lost packets. It uses acknowledgments to confirm the receipt of messages sent by the sender, and negative acknowledgments to confirm the loss of messages sent by the sender. It consists of *aSnd* fraction at the sender and *aRcv* fraction at the receiver. The *aSnd* 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 receiver. When the *aRcv* fraction at the receiver receives a packet out of order, it waits for few more packets before sending a negative acknowledgment to the sender. When all packets in a group are received, it sends an acknowledgment for that group to the sender.

The reactive component provides tolerance to message loss faults $F2$ shown in Fig. 14. Faults of class $F2$ causes loss of messages from the channel. For simplicity, we assume that acknowledgment messages are not lost; however, the component can be easily

Component *ack*

Fraction *aSnd*

inp sQ : **queue** of **integer**
$r1, r2$

var $n, w, g_i, p_i, g_a, g_{na}, p_{na}, m$: **integer** {initially, $p_i = g_i = 0$}
$sQcopy_i$: **queue** of **integer** {initially, Empty}
snt_i : **array** $[0..w-1, 0..n-1]$ of **integer** {initially $\perp$}

param i : $i = 1, 2$

begin

copy$_i$: isEmpty$(sQcopy_i)$ $\rightarrow$ $sQcopy_i := sQ$

[] **send**$_i$: $\neg$isEmpty$(sQcopy_i) \wedge snt_i[g_i, p_i] = \perp$ $\rightarrow$ $snt_i[g_i, p_i] := \text{data}(g_i, p_i, \texttt{head}(sQcopy_i))$;
$C_{s,r_i} := C_{s,r_i} \circ snt_i[g_i, p_i]$;
$p_i := (p_i + 1) \bmod n$;
if $p_i = 0$ **then**
$g_i := (g_i + 1) \bmod w$
fi

[] **resend**$_i$: $\texttt{type}(C_{r_i,s}) = \text{nack}$ $\rightarrow$ $g_{na}, p_{na}, m := \texttt{head}(C_{r_i,s})$;
if $snt_i[g_{na}, p_{na}] \neq \perp$ **then**
send $snt_i[g_{na}, p_{na}]$ to r_i
fi

[] **ack_rcv**$_i$: $\texttt{type}(C_{r_i,s}) = \text{ack}$ $\rightarrow$ $g_a, snt_i[g_a, 0..n-1] := \texttt{head}(C_{r_i,s}), \perp$

end

Fraction *aRcv*$_i$

inp rQ
s

var $n, w, g, p, k, nxt\_grp, m$: **integer** {initially $k = nxt\_grp = 0$}
$rbufQ$: **array** $[0..w-1, 0..n-1]$ of **integer** {initially $\perp$}
ud_grp : **array** $[0..w-1]$ of **boolean** {initially *false*}

param j : $0 \leq i \leq w-1$

begin

receive : $\neg$isEmpty(C_{s,r_i}) $\rightarrow$ $g, p, m := \texttt{head}(C_{s,r_i})$;
$rbufQ[g, p], ud\_grp[g] := m, true$

[] **deliver**$_j$: $ud\_grp[j] = true$ $\rightarrow$ **if** $\texttt{count}(rbufQ[j, 0..n-1] = \perp) = n$ **then**
$rQ := rQ \circ rbufQ[j, 0..n-1]$;
$rbufQ[j, 0..n-1], C_{r_i,s} := \perp, C_{r_i,s} \circ \text{ack}(j)$;
$ud\_grp[j], nxt\_grp := false, (j+1) \bmod w$
fi

[] **send_nack** : $\texttt{count}(ud\_grp[0..w-1] = true) > 2$ $\rightarrow$ **for** $k = 0$ **to** $n-1$
if $rbufQ[nxt\_grp, k] = \perp$ **then**
$C_{r_i,s} := C_{r_i,s} \circ \text{nack}(nxt\_grp, k)$
fi
end

end

Fig. 13. Acknowledgment component.

extended to deal with faults that lose acknowledgments by using *timeout* guards.

msg_loss$_i$: $m \in C_{s,r_i}$ $\rightarrow$ $C_{s,r_i} := C_{s,r_i} - \{m\}$

Fig. 14. Fault class $F2$

Fig. 13 shows the abstract version of the reactive component. The *aSnd* fraction consists of four actions: `copy`, `send`, `resend`, `ack_rcv`. The *aRcv* fraction consists of three actions: `receive`, `deliver`, and `send_nack`. These fractions are composed with processes that will use them. The message communication program composed with the reactive component is shown in Fig. 15.

Program $\mathcal{P}_{ack}$
process s
  var : $\mathcal{P}_{aa-fec}.s.\text{var} \cup ack.aSnd.\text{var}$
  param $i : i = 1, 2$
  begin
      ack.aSnd.copy$_i$
    [] **ack.aSnd.send**$_i$
    [] **ack.aSnd.send_again**$_i$
    [] **ack.aSnd.ack_rcv**$_i$
  end
process $r_i[i = 1, 2]$
  begin
  var : $\mathcal{P}_{aa-fec}.r_i.\text{var} \cup ack.aRcv_i.\text{var}$
  param $k : 0 \leq k \leq w - 1$
  begin
      ack.aRcv.receive
    [] **ack.aRcv.deliver**$_k$
    [] **ack.aRcv.send_nack**
  end

Fig. 15. Message communication program (with reactive component).

Specification of program using the reactive component. Program using the reactive component satisfies the same specification as the communication program (cf. Sect. 4.2). Additionally, it tolerates message loss faults $F2$.

Invariant. The invariant of the program using the reactive component is $S_R = S_1 \wedge S_A$, where

$S_A = \forall i : m_i \in mQ \Rightarrow ((m_i \in sQcopy_1 \veebar m_i \in r_1.rQ \veebar (m_i \notin (sQcopy_1 \cup r_1.rQ) \Rightarrow (m_i \in snt_1 \wedge (m_i \in C_{s,r_1} \veebar m_i \in r_1.rbufQ)))) \wedge (m_i \in sQcopy_2 \veebar m_i \in r_2.rQ \veebar (m_i \notin (sQcopy_2 \cup r_2.rQ) \Rightarrow (m_i \in snt_2 \wedge (m_i \in C_{s,r_2} \veebar m_i \in r_2.rbufQ))))))$.

In the above invariant, S_A indicates that for a message m, exactly one of the following is true:

- m is at the sender, and is not yet sent
- m is received by the receiver
- m is buffered by the sender, and m is either in the channel or is buffered at the receiver

Fault-span. The $F2$-span of the program using the reactive component is $T_R = S_1 \wedge T_A$, where

$T_A = \forall i : m_i \in mQ \Rightarrow ((m_i \in sQcopy_1 \veebar m_i \in r_1.rQ \veebar (m_i \notin (sQcopy_1 \cup r_1.rQ) \Rightarrow m_i \in snt_1)) \wedge (m_i \in sQcopy_2 \veebar m_i \in r_2.rQ \veebar (m_i \notin (sQcopy_2 \cup r_2.rQ) \Rightarrow m_i \in snt_2)))$.

In the above fault-span, T_A indicates that a message is either not yet sent by the sender, or is received by the receiver, or if it sent by the sender but not yet received by the

receiver then it is buffered by the sender.

6.3 Adaptation: Replacement of the proactive with the reactive component

Given a program shown in Fig. 11, the adaptation of replacing the proactive component with the reactive component converts the program to one shown in Fig. 15.

Specification during adaptation. The specification during adaptation is that S_1 continues to be true during adaptation in presence of faults F_1.

Program $\mathcal{P}_{aa\text{-}ip_1}$
process s
  var : $\mathcal{P}_{aa-fec}.s.$var
  begin
      fec.encoder.fec_send
    [] $aa_2 : aa_1 \wedge l = u \ \rightarrow \ transformTo(\mathcal{P}_{aa\text{-}ip_2}.s, \Phi_{aa_2})$;
  end
process $r_i[i = 1, 2]$: same as in Fig. 11

Fig. 16. Intermediate program $\mathcal{P}_{aa\text{-}ip_1}$.

We identify the intermediate programs during adaptation after each atomic adaptation. The execution of adaptive action aa_1 in $\mathcal{P}_{aa\text{-}fec}$ results in the intermediate program $\mathcal{P}_{aa\text{-}ip_1}$ shown in Fig. 16. $\mathcal{P}_{aa\text{-}ip_1}$ does not encode more packets, but will send any remaining encoded packets. In the execution of $\mathcal{P}_{aa\text{-}ip_1}$, eventually all the encoded packets are sent to the receivers. Thus, the guard of adaptive action aa_2 becomes true. The transitional-invariant of $\mathcal{P}_{aa\text{-}ip_1}$ is: $TS_5 = S_Q \wedge S_6$, where S_Q is defined earlier in Sect. 4.3, and

$S_6 = (\forall j : j \geq u : encQ[j, 0..n-1] = \perp) \wedge (l \leq u)$.

The transitional-faultspan TT_5 of $\mathcal{P}_{aa\text{-}ip_1}$ is same as TS_5.

Program $\mathcal{P}_{aa\text{-}ip_2}$
process s
  var : $\mathcal{P}_{aa-fec}.s.$var
  begin
    $aa_5 : aa_3 \wedge aa_4 \ \rightarrow \ transformTo(\mathcal{P}_{ack}.s, \Phi_{aa_5})$;
  end
process $r_i[i = 1, 2]$: same as in Fig. 11

Fig. 17. Intermediate program $\mathcal{P}_{aa\text{-}ip_2}$.

The execution of aa_2 results in the intermediate program $\mathcal{P}_{aa\text{-}ip_2}$ shown in Fig. 17. $\mathcal{P}_{aa\text{-}ip_2}$ does not send any packets, but the packets that are there in the channel can still be received by the receivers r_1 and r_2. In the execution of $\mathcal{P}_{aa\text{-}ip_2}$ eventually all the packets in the channel are read and no new packets are added in the channel from sender to receiver. Thus, the guards of the adaptive actions aa_3 and aa_4 eventually get enabled.

The transitional-invariant of $\mathcal{P}_{aa\text{-}ip_2}$ is: $TS_6 = S_1 \wedge S_7 \wedge S_8$, where S_1 is defined earlier in Sect. 4.2, and

$S_7 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \ \veebar \ ((m_i \in r_1.rQ \ \veebar \ m_i \in \text{data}(C_{s,r_1} \cup r_1.rbufQ)) \ \wedge \ (m_i \in r_2.rQ \ \veebar \ m_i \in \text{data}(C_{s,r_2} \cup r_2.rbufQ))))$, and

$S_8 = (\forall j : j \geq u : encQ[j, 0..n-1] = \perp) \wedge (l = u)$.

The transitional-faultspan TT_6 of $\mathcal{P}_{aa\text{-}ip_2}$ is same as TS_6.

Program $\mathcal{P}_{aa\text{-}ip_3}$
process s : same as in Fig. 17
process r_1 : same as in Fig. 15
process r_2 : same as in Fig. 11

Fig. 18. Intermediate program $\mathcal{P}_{aa\text{-}ip_3}$.

Since aa_3 and aa_4 occur independently, we consider both possible orderings between them. The execution of adaptive action aa_2 in $\mathcal{P}_{aa\text{-}ip_2}$ results in the intermediate program $\mathcal{P}_{aa\text{-}ip_3}$ shown in Fig. 18. In $\mathcal{P}_{aa\text{-}ip_3}$, receiver r_1 has replaced its fraction, whereas receiver r_2 has not yet replaced its fraction and can receive any remaining packets in the channel from s to r_2. Eventually, in the execution of $\mathcal{P}_{aa\text{-}ip_3}$ the guard of adaptive action aa_3 gets enabled and aa_3 is executed resulting in the intermediate program $\mathcal{P}_{aa\text{-}ip_5}$. The transitional-invariant of $\mathcal{P}_{aa\text{-}ip_3}$ is $TS_7 = S_1 \wedge S_8 \wedge S_9 \wedge S_{10}$, where

$S_9 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \ \veebar \ (m_i \in r_1.rQ \ \wedge \ (m_i \in r_2.rQ \ \veebar \ m_i \in \text{data}(C_{s,r_2} \cup r_2.rbufQ))))$, and

$S_{10} = \text{isEmpty}(C_{s,r_1}) = true \wedge r_1.rbufQ = \perp$.

The transitional-faultspan TT_7 of $\mathcal{P}_{aa\text{-}ip_3}$ is same as TS_7.

Program $\mathcal{P}_{aa\text{-}ip_4}$
process s : same as in Fig. 17
process r_1 : same as in Fig. 11
process r_2 : same as in Fig. 15

Fig. 19. Intermediate program $\mathcal{P}_{aa\text{-}ip_4}$.

The execution of adaptive action aa_4 in $\mathcal{P}_{aa\text{-}ip_2}$ results in the intermediate program $\mathcal{P}_{aa\text{-}ip_4}$ shown in Fig. 19. In $\mathcal{P}_{aa\text{-}ip_4}$, receiver r_2 has replaced its fraction, whereas receiver r_1 has not yet replaced its fraction and can receive any remaining packets in the channel from s to r_1. Eventually, in the execution of $\mathcal{P}_{aa\text{-}ip_4}$ the guard of adaptive action aa_3 gets enabled and aa_3 is executed resulting in intermediate program $\mathcal{P}_{aa\text{-}ip_5}$. The transitional-invariant of $\mathcal{P}_{aa\text{-}ip_4}$ is $TS_8 = S_1 \wedge S_8 \wedge S_{11} \wedge S_{12}$, where

$S_{11} = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \ \veebar \ ((m_i \in r_1.rQ \ \veebar \ m_i \in \text{data}(C_{s,r_1} \cup r_1.rbufQ)) \ \wedge \ m_i \in$

$r_2.rQ)$), and

$S_{12} = \text{isEmpty}(C_{s,r_2}) = true \wedge r_2.rbufQ = \perp$.

The transitional-faultspan TT_8 of $\mathcal{P}_{aa\text{-}ip_4}$ is same as TS_8.

Program $\mathcal{P}_{aa\text{-}ip_5}$

process s : same as in Fig. 17

process $r_i[i = 1, 2]$: same as in Fig. 15

Fig. 20. Intermediate program $\mathcal{P}_{aa\text{-}ip_5}$.

In the intermediate program $\mathcal{P}_{aa\text{-}ip_5}$ shown in Fig. 20, only the adaptive action aa_5 is enabled, and execution of aa_5 results in the new program $\mathcal{P}_{ack}$. The transitional-invariant of $\mathcal{P}_{aa\text{-}ip_5}$ is $TS_9 = S_1 \wedge S_{10} \wedge S_{12} \wedge S_{13}$, where

$S_{13} = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \ \veebar \ (m_i \in r_1.rQ \wedge m_i \in r_2.rQ))$.

The transitional-faultspan TT_9 of $\mathcal{P}_{aa\text{-}ip_5}$ is same as TS_9.

State mapping. The state mapping for each adaptive action is shown in Table 2. Each adaptive action initializes the state of the new process when it is executed based on this mapping.

Mapping Function	Process Affected	New State
Φ_{aa_1}	s	Identity mapping
Φ_{aa_2}	s	Identity mapping
Φ_{aa_3}	r_1	$\{rQ, s\}$ - Identity mapping, $V(r_1) - \{rQ, s\}$ - Initial mapping
Φ_{aa_4}	r_2	$\{rQ, s\}$ - Identity mapping, $V(r_2) - \{rQ, s\}$ - Initial mapping
Φ_{aa_5}	s	$\{sQ, r_1, r_2\}$ - Identity mapping, $V(s) - \{sQ, r_1, r_2\}$ - Initial mapping

Table 2. State mapping for each adaptive action.

Fig. 21 shows the adaptation lattice for the adaptation of adding the proactive component.

Theorem 2.1. *The adaptation lattice of Fig. 21 is a transitional-faultspan (F1-span) lattice for the adaptation of replacing the proactive component with the reactive component. Hence, the adaptation is correct in presence of faults.* □

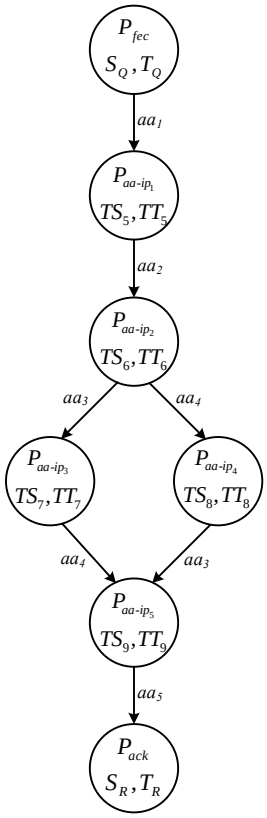


Fig. 21. Adaptation lattice for replacement of proactive component with reactive component.

7 Implementation Framework for Correct Adaptation

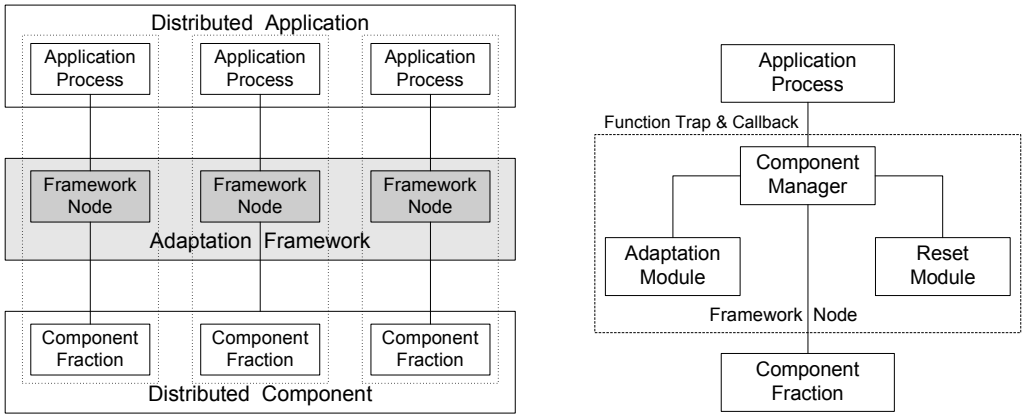
In the previous sections, we have discussed how to model an adaptation in a distributed system and presented the approach based on the adaptation lattice to verify the adaptation. However, one potential problem with the use of such lattice is that it may be expensive to perform all the verification tasks identified by the lattice. In [37], we have showed how the size of the lattice increases with the number of atomic adaptations and the number of processes, and have identified the tradeoffs associated with the size of the lattice.

Even in cases where verifying the conditions of the transitional-invariant lattice is difficult, the lattice is valuable in the context of testing. In particular, the transitional-invariants can be utilized to determine if a particular execution of the dynamic adaptation satisfies the required constraints specified by the transitional-invariant lattice. In [38], we have shown how we can perform testing of dynamic adaptation using predicate detection techniques. Also, we may consider verifying/testing a subset of the tasks identified by the adaptation lattice. For example, we may choose to verify only subset of adaptation paths in the lattice and ensure that the adaptation follows only one of the verified paths, or we may choose to only verify that the adaptation follows some

path in the lattice, and that the program after adaptation starts in its invariant. Additionally, we may choose to verify the transitional-invariants at certain instances during adaptation, such as at the start of each intermediate program, rather than during the entire execution of the intermediate program.

In general, while transforming the adaptation model into its concrete implementation, we need to ensure that what has been verified for the model holds for the corresponding implementation also. With this motivation, in this section, we give a brief description of an adaptation framework that allows the developer to implement the adaptation so that it follows some adaptation path in the lattice. Also, the framework provides an execution trace of the adaptation that can be used for testing. Additional details of this framework are available in [39]. We are currently extending this framework to verify the transitional-invariants for the intermediate programs along the adaptation path.

The framework is as shown in Fig. 22(b) and is based on *distributed reset protocol* [40]. A *framework node* is instantiated at each process in the application (cf. Fig. 22(a)). Each framework node consists of a *component manager*, an *adaptation module* and a *reset module* (cf. Fig. 22(b)). The component manager performs the addition, removal and replacement of component fractions. The adaptation module selects an appropriate component, the choice of which is orthogonal to the design of the framework. The reset module ensures that the adaptation follows the desired adaptation path.



(a) Adaptation Framework (b) Framework Node

Fig. 22. Framework for Implementing Correct Adaptation

As discussed in Sect. 2, each adaptive action in the lattice has a guard associated with it, and the adaptive action is executed only when its guard is true. For adaptation to follow a path in the lattice, the component should provide a function to check for its state. To perform this check, the framework requires the developer to provide *checkState* function for each fraction of the component. We consider following adaptive actions in our work:

(*i*) `block` a fraction, (*ii*) `add` a fraction, (*iii*) `remove` a fraction, and (*iv*) `replace` a fraction. Corresponding to the adaptive actions, the *checkState* function returns one of the five values: (*i*) *safetoblock*, (*ii*) *safetoadd*, (*iii*) *safetoremove*, (*iv*) *safetoreplace*, and (*v*) *unsafe*. The return value is computed based on the guards and the adaptive action that needs to be executed at the given fraction. The framework automatically invokes this function as needed, and based on the return value it executes the appropriate adaptive action. In the next subsection, we show how the framework uses the *checkState* function to perform the adaptation.

7.1 Reset-Based Composition of Distributed Component

In this subsection, we discuss replacement of a distributed component; addition and removal being special cases of replacement. We show how our framework uses the *distributed reset protocol* to perform this adaptation.

For the discussion of component replacement, assume that the adaptation module at a process, say X, has decided to replace the distributed component. We call X the *initiator*. To replace the component, the component manager at X generates a *magic number* for the instance of the new component. The magic number is generated by using the initiator ID, the current time at the initiator, and is used to uniquely identify the instance of the new component. The component manager appends the magic number of the component that the application is using in the message header while communicating with component managers at other processes of the application. The component manager at X uses the reset module for changing the distributed component.

Overview of steps in dynamic component replacement. The dynamic component replacement is achieved by three waves, namely, an *initialization wave*, a *transition wave* and a *completion wave*. The reset module at X initiates the component replacement by sending the initialization wave. In the initialization wave, all processes change to the *transit state* and initialize the component fraction of the new distributed component. Thus, in the transit state, a process has initialized the new component fraction, although it is still using the old component fraction. Upon successful completion of the initialization wave, the reset module starts the transition wave. Each process receiving the transition wave invokes the *checkState* function of the component fraction to determine the state of the component fraction. During the transition wave, the processes remove the old component fraction and add the new component fraction depending on the state information returned by the *checkState* function. The *checkState* function is designed to be non-blocking. If the *checkState* function returns *unsafe*, then it is called periodically until it returns a value corresponding to some adaptive action. After a leaf

process has completed the replacement of its component fraction, it sends the completion wave to its parent. Further, if a non-leaf process has completed the replacement of its component fraction and it has received the completion wave from all of its children, it propagates the completion wave to its parent. The completion wave eventually reaches the initiator X. We refer readers to [39] for details on these reset waves.

In above discussion, we used two complete waves (initialization and transition) and one half wave (completion wave) for the adaptation. However, based on the adaptation specification the framework can be customized to use less number of waves. For example in case of mixed-mode adaptation, it is possible to customize the framework to perform adaptation using only one half wave.

8 Discussion

In this section, we discuss some of the issues related to our approach.

Why is the specification during adaptation a safety specification?

The specification of the application before adaptation can be arbitrary. However, during adaptation the specification should be a safety specification. It is not desirable to delay the adaptation to satisfy liveness during adaptation. Rather, we would expect the adaptation to complete as quickly as possible and the new program to satisfy the safety and liveness after adaptation. For example, consider a transaction processing system with liveness guarantees to commit or abort. In this case, either the adaptation should not start in the middle of the transaction, or if the adaptation can be started in the middle of the transaction than the liveness should be met once the adaptation completes. Thus, the implicit liveness specification during adaptation is that adaptation completes.

What is the tradeoff between concurrency among atomic adaptations and verification complexity of adaptation?

In an adaptation, the atomic adaptations occurring at different processes may occur in an asynchronous manner. Therefore, to verify a given adaptation, we need to consider all possible orderings of concurrent atomic adaptations. Putting concurrent atomic adaptations in various possible orderings is a potential cause of the explosion in size of the adaptation lattice. Clearly, the complexity of verifying adaptation depends on the size (number of nodes and edges) of the adaptation lattice, as each node and each edge in the lattice needs to be verified independently. In [37], we discuss the tradeoff between concurrency during adaptation and complexity of verifying that adaptation.

Based on the tradeoff between concurrency of adaptation and complexity of verifying that adaptation, we can choose a subgraph (sublattice) of the given lattice that has all the properties of the adaptation lattice as defined in Sect. 3. The adaptation in this case has to be constrained so that it follows the path in the sublattice.

How is the transitional-invariant lattice constructed?

While details of finding these predicates is outside the scope of the paper, we note that the techniques developed to calculate invariants (e.g. [28, 41]) can also be used to find transitional-invariants. For a given adaptation model, we can perform reachability analysis for each intermediate program obtained after execution of the atomic adaptation. The reachability computation for each intermediate program helps in identifying the transitional-invariants for that intermediate program, and we can construct a transitional-invariant lattice for the given adaptation. Furthermore, the techniques for dynamically discovering likely invariants from execution of the system such as [42] can be used to find transitional-invariants for the adaptation lattice.

How is the checkState *function defined?*

Given the adaptation lattice, the guards of the atomic adaptation, and the information about the previous atomic adaptations on the the path can be used to compute the *checkState* function. Further, in case the adaptation lattice is not available, the *checkState* function can be calculated based on the *dependency analysis* as discussed in [4]. The dependency analysis basically identifies different kinds of dependency relations that exist among the fractions of the component and between the fractions and the application process where it is installed. Also, this function could be defined with the help of *safe states* considered in [4, 43].

Will the adaptation block forever if any fraction is not available for addition or replacement?

It is desirable not to start an adaptation rather than having adaptation start and block indefinitely. The initialization wave in the distributed reset ensures that all fractions are available and can be initialized before it begins the replacement (or addition) of component fractions. This guarantees that once the adaptation has started, it will not block indefinitely due to unavailability of any fractions.

9 Conclusion

In this paper, we presented an approach to verify the correctness of adaptation. We introduced the notion of transitional-invariant lattice and transitional-faultspan lattice to verify the correctness of adaptation in absence and presence of faults, respectively. We demonstrated the use of our approach in verifying two example adaptations: (*i*) adding the proactive component to a message communication application in absence of faults, and (*ii*) replacing the proactive component with the reactive component in presence of faults. We also described an architecture framework that we used to implement the adaptation. For reasons of space, we refer readers to [39,44], where we discuss additional examples.

In our approach for verifying adaptation, there are two main parts: (1) identifying the transitional-invariants (respectively, transitional fault-spans), and (2) verifying that they meet the properties of the transitional-invariant (respectively, transitional fault-span) lattice. The latter part is a traditional problem considered in program verification and, hence, existing techniques can be used in this part. In Sect. 8, we identified how existing approaches could be used for the former part as well. However, methods for identifying (either automatically or semi-automatically) transitional-invariants and transitional fault-spans are a future work that is outside the scope of this paper.

We also briefly described a framework that can be used in implementing dynamic adaptation (additional details of this framework are available in [39]). Currently the framework allows the adaptation to follow a path that is defined by the *checkState* function (which is itself determined from the lattice). We plan to extend it so that it can also *test* that the appropriate transitional-invariants (respectively, transitional fault-spans) are also satisfied. This framework is also applicable where the designer can identify the *structure* of the transitional-invariant lattice (typically, a easy task) but cannot identify the transitional-invariants (typically, a challenging task). Also, we have also used this approach to provide a tradeoff between concurrency in adaptation and the cost of its verification [37].

There are several possible extensions to this work. One extension to this work is to generate the lattices automatically, given the invariants (and, fault-spans) of the application before adaptation and after adaptation. Based on the adaptation requirements, the automatic generation of the lattices would enable us to identify different paths to achieve adaptation and also ensure correctness of those paths.

Also, in our framework, currently the *checkState* function is written by the adaptation manager or component developer. It is desirable to automate the development of

checkState function. Given the transitional-invariant lattice, it is possible to design an algorithm to define the *checkState* function for each fraction. We are exploring ways to automatically write the *checkState* function with minimum possible human intervention.

References

[1] Sandeep Kulkarni and Karun Biyani. Correctness of component-based adaptation. In *International Symposium on Component-based Software Engineering - CBSE, at ICSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 48–58, May 2004.

[2] 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.

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

[4] Sandeep S. Kulkarni, Karun N. Biyani, and Umamaheswaran Arumugam. Composing distributed fault-tolerance components. In *Workshop on Principles of Dependable Systems - PoDSy, at DSN*, pages W127–136, June 2003.

[5] J. Hallstrom, W. Leal, and A. Arora. Scalable evolution of highly available systems. *Transactions of the Institute for Electronics, Information and Communication Engineers*, E86-D(10):2154–2164, 2003.

[6] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behavior. In *ECOOP*, pages 205–230, 2002.

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

[8] Jean-Charles Fabre and Tanguy Perennou. FRIENDS: A flexible architecture for implementing fault tolerant and secure distributed applications. In *European Dependable Computing Conference*, pages 3–20, 1996.

[9] Ralph Keller and Urs Hölzle. Binary component adaptation. *Lecture Notes in Computer Science*, 1445, 1998.

[10] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[11] Karun Biyani and Sandeep Kulkarni. Issues in mixed-mode adaptation. Technical Report MSU-CSE-07-6, Michigan State University, Jan 2007.

[12] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *International Workshop on Self-Managed Systems (WOSS)*, 2004.

[13] D. L. Metayer. Describing software architecture styles using graph grammers. *IEEE Transaction on Software Engineering*, 24(7):521–533, 1998.

[14] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation*, volume 1764 of *LNCS*. Springer, 1998.

[15] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A graph based architectural (re)configuration language. In *Proceedings of the 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundatations of Software Engineering (ESEC/FSE 2001)*, volume 26 of *Software Engineering Notes*, pages 21–32, 2001.

[16] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of Conference on Fundamental Approaches to Software Engineering*, volume LNCS 1382, pages 21–35, 1998.

[17] C. E. Cuesta, P. de la Fuente, and M. Barrio-Solarzano. Dynamic coordination architecture through use of reflection. In *Proceedings of ACM Symposium on Applied Computing*, pages 134–140. ACM Press, 2001.

[18] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, pages 177–186, 1998.

[19] J. Kramer, J. Magee, and M. Sloman. Configuring distributed systems. In *Proceedings of the 5th Workshop on ACM SIGOPS European Workshop*, pages 1–5. ACM Press, 1992.

[20] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, and J. 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.

[21] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE: Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 287–296, Helsinki, Finland, September 2003.

[22] Leonardo Mariani and Mauro Pezzè. A technique for verifying component-based software. In *International Workshop on Test and Analysis of Component Based Systems*, pages 17–30, Barcelona, Spain, March 27–28, 2004.

[23] Sagar Chaki, Natasha Sharygina, and Nishant Sinha. Verification of evolving software. In *3rd International Workshop on Specification and Verification of Component-based Systems*, pages 55–61, 2004.

[24] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software - Practice and Experience*, 23(9):949–964, 1993.

[25] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985.

[26] Sandeep Kulkarni. *Component-based Design of Fault Tolerance*. PhD thesis, Ohio State University, 1999.

[27] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. In *ACM Transactions on Programming Languages and Systems*, volume 4, pages 455–495, July 1982.

[28] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[29] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 1993.

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

[31] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE transactions on Software Engineering*, 24(1):63–78, January 1998.

[32] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[33] A. Arora and Sandeep S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, 24(6):435–450, 1998.

[34] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[35] G. Varghese. *Self-stabilization by local checking and correction*. PhD thesis, MIT/LCS/TR-583, 1993.

[36] M. G. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17:911–921, 1991.

[37] Karun Biyani and Sandeep Kulkarni. Concurrency tradeoffs in dynamic adaptation. In *International Workshop on Assurance in Distributed Systems and Networks - ADSN, at ICDCS*, July 2006.

[38] Karun Biyani and Sandeep Kulkarni. Testing dynamic adaptation in distributed systems. Technical Report MSU-CSE-07-1, Michigan State University, Jan 2007.

[39] Karun Biyani. Dynamic composition of distributed components. Master's thesis, Michigan State University, Available at: `http://www.cse.msu.edu/~biyanika/thesis`, 2003.

[40] A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[41] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[42] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, (Seattle, Washington), 2000.

[43] 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.

[44] Sandeep Kulkarni and Karun Biyani. Correctness of component-based adaptation. Technical Report MSU-CSE-04-2, Department of Computer Science, Michigan State University, January 2004.