Concurrency and Complexity in Verifying Dynamic Adaptation: A Case Study [⋆]

Karun N. Biyani[⋆⋆] Sandeep S. Kulkarni[⋆⋆⋆]

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

Abstract. Software systems need to adapt as requirements change, environment conditions vary, and bugs are discovered and fixed. In systems that need to provide continuous operation, it is important that the adaptation be done with minimal interruption in the execution of the system. In context of verification of these adaptive systems, the verification needs to be done for the system before adaptation, for the system during adaptation, and for the system after adaptation. While the existing techniques of program verification can be directly applied to verify the system before and the system after adaptation, they cannot be applied directly to verify the system during adaptation. This is because the system during adaptation is not well-defined as it consists of parts of both the old system (system before adaptation) and the new system (system after adaptation). In our previous work, we presented an approach based on *adaptation lattice* for verifying the correctness of dynamic adaptation. The complexity of our approach 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 this paper, we discuss the tradeoff between concurrency of adaptation and complexity of verifying that adaptation.

1 Introduction

Software systems evolve or adapt to modify their functional or non-functional behavior. This adaptation is required because of changes in requirements and/or environment conditions. The adaptation may reconfigure system parameters, or add, remove or replace software components. Such adaptation can be done either statically (during compile-time or load-time) or dynamically (during run-time). In systems that are required to provide continuous (uninterrupted) service, the adaptation needs to be done dynamically. In this paper, we focus on the verification of such dynamic adaptive systems. (For brevity, unless mentioned otherwise, in this paper, we use adaptation to denote dynamic adaptation.)

The issues in adaptation can be classified into two broad categories: syntactic and semantic. Syntactic issues in adaptation deal with mechanisms of adaptation, such as, parameter checking, interface compatibility, dynamic loading, and reflection. Semantic issues address the correctness aspect of adaptation, such as,

[⋆] 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.

[⋆⋆] Email: `biyanika@cse.msu.edu`

[⋆⋆⋆] Email: `sandeep@cse.msu.edu`, Phone: +1-517-355-2387

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **2005**	2. REPORT TYPE	3. DATES COVERED **00-00-2005 to 00-00-2005**
4. TITLE AND SUBTITLE **Concurrency and Complexity in Verifying Dynamic Adaptation: A Case Study**		5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Michigan State University,Department of Computer Science and Engineering,East Lansing,MI,48824**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S) 11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**15**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

verifying the system during adaptation and the system after adaptation. Several approaches (e.g. [1–6]) addressing adaptation in distributed systems have been discussed in literature. However, these approaches mostly address the syntactic issues in adaptation.

Since current literature does not sufficiently address the semantic issues in adaptation, in [7], we focused on an approach for verifying adaptation in a distributed system. Since such adaptation involves modification to several (if not all) processes involved in the system, it cannot be performed atomically. We, therefore, introduced the notion of *atomic adaptation*; typically only a single process will be involved in an atomic adaptation. Thus, the adaptation in a distributed system can be modeled as a sequence of several atomic adaptations. It follows that during adaptation, the system may consist of parts of the old program (i.e., program before adaptation) and parts of the new program (i.e., program after adaptation). Hence, to guarantee correctness of adaptation, in [7], we focus on the properties of such *intermediate programs*.

To verify the properties of an adaptive program, in [7], we introduced an *adaptation lattice*. Specifically this lattice enables us to verify properties of adaptation by verifying the (safety) properties of these intermediate programs. Each node in the lattice corresponds to an intermediate program and needs to be verified independently. Further, each edge in the lattice also needs to be verified. Thus, the complexity of verification depends on the size of the adaptation lattice.

In this paper, we investigate approaches for reducing the size of the adaptation lattice and, thereby reduce the cost of verification of adaptation. Specifically, we evaluate the tradeoff between concurrency of adaptation and the verification complexity of that adaptation. We show that such a tradeoff — that is previously considered in the context of concurrency and verification of single non-adaptive programs — can be extended in the context of adaptive systems. We illustrate this tradeoff using the case study in which the adaptation replaces a centralized mutual exclusion protocol ([8]) with a distributed mutual exclusion protocol ([9]).

Organization of the paper. In Section 2, we discuss how distributed programs and adaptations are modeled. We describe the adaptation lattice used in verification of adaptation in Section 3. The tradeoff in verifying adaptation is discussed in Section 4. We present our case study in Section 5. In Section 6, we discuss some issues related to our work, and finally conclude in Section 7.

2 Modeling Adaptation

In this section, we describe how we model the adaptive programs in order to show their correctness. This modeling is based on our previous work in [7]. We consider adaptations where the program needs to add, remove or replace distributed protocols. A protocol implements a part of the desired behavior of the system. It consists of one of more *fractions* [1]. Each fraction is associated with one process of the program. For example, consider a protocol that provides secure communication between a sender and a receiver. Such a protocol consists of two fractions, namely, *encryption fraction* at the sender that encrypts the packets before sending and *decryption fraction* at the receiver that decrypts the encrypted packets received from the sender. For simplicity, we consider only one fraction at each process, although, our approach can be used even if multiple fractions are associated at a process.

Atomic Adaptation. To add a protocol to a distributed program, each fraction of the protocol needs to be added at processes of the program. Similarly, to remove a protocol, each fraction of the component needs to be removed from the corresponding process of the program. Thus, an adaptation in distributed system involves changes at various processes of the system. In other words, adaptation in distributed programs involves multiple steps. The key to verifying adaptation in distributed programs is identifying individual atomic steps in adaptation. We divide this multi-step adaptation into multiple *atomic adaptations* each occurring at only one process. Thus, an adaptation consists of a sequence of atomic adaptations. Each atomic adaptation a special type of *action* (defined later in this section) and is represented by a name and has a *guard* (defined later in this section). An atomic adaptation is performed when the guard corresponding to it becomes true.

Intermediate Program. We define a system before adaptation as *old program* and a system after adaptation as *new program.* Each atomic adaptation modifies the system into an *intermediate 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.

In the case study discussed in Section 5, we consider adaptation that performs replacement of protocols. Addition and removal of protocols are considered as special case of replacement (cf. Section 6).

Now, we discuss how we model such distributed programs and the multi-step adaptation performed in them. We note that general purpose languages such as C++/Java that are used in existing adaptation techniques (e.g., [1–4, 10]) are not suitable for our task as verification of programs in these languages is difficult even in the absence of adaptation. For this reason, while proving correctness of programs, we often consider their abstract version where several implementation details are omitted. We use a variation of the guarded commands [11] for our purpose. We note that there are techniques [12] that allow one to obtain a guarded command representation from a program in general purpose language such as C. Also, there are techniques [13,14] that allow one to transform a program in guarded commands into a program in general purpose languages. We describe the modeling of the distributed programs and their adaptation, next.

2.1 Modeling Adaptive Program

Program and Process. A program $\mathcal{P}$ is specified by a set of global constants, a set of global variables and a finite set of processes. A process p is specified by a set of local constants, a set of local variables and a finite set of *actions* (defined later in this section). Variables declared in $\mathcal{P}$ can be read and written by the actions of all processes. The processes in a program communicate with one another by sending and receiving messages over unbounded channels that connect the processes. We use the notation $ch.p.q$ to denote the content of the channel from process p to process q, and $\#ch.p.q$ to denote the number of messages in the channel from p to q.

Protocol and its Fractions. A protocol is a set of global constants, a set of global variables, and a finite set of fractions that are involved in providing a common functionality. A fraction is a set of local constants, a set of local variables, and a finite set of actions that are associated with a single process. A protocol and a fraction also has input parameters, which are values of the variables supplied by the program; and output parameters, which are values returned to the program.
State and State Predicate. A *state of process* p is defined by a value for each variable of p, chosen from the predefined domain of the variable. A *state of program* $\mathcal{P}$ is defined by a value for each global variable of $\mathcal{P}$, the state of all its processes and the contents of all channels. A state predicate of p is a boolean expression over the constants and variables of p. A state predicate of $\mathcal{P}$ is a boolean expression over constants and variables of all processes, all global constants and global variables, and the contents of all channels. A state and a state predicate of a protocol and a fraction are defined accordingly.
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 is a combination of one or more of the following: a state predicate of p, a receiving guard of p, or a timeout guard. A receiving guard of p is of the form `rcv` $\langle message \rangle$ `from` $\langle q \rangle$. A timeout guard is of the form `timeout` $\langle state\ predicate\ of\ \mathcal{P} \rangle$. The statement of an action updates zero or more variables and/or sends one or more messages. An action can be executed only if its guard evaluates to true. To execute an action, the statement of that action is executed atomically. A sending statement of p is of the form `send` $\langle message \rangle$ `to` $\langle q_1, ..., q_n \rangle$, which sends a message to one or more processes . We say that an action of p is enabled in a state of $\mathcal{P}$ iff its guard evaluates to true in that state.
Computation. A computation of $\mathcal{P}$ is a sequence of states $s_0, s_1, ...$ such that for each j, $j > 0$, s_j is obtained from state s_{j-1} by executing an action of $\mathcal{P}$ that is enabled in the state s_{j-1}, and satisfies the following conditions: (*i*) *Non-determinism*: Any action that is enabled in a state of $\mathcal{P}$ can be selected for execution at that state, (*ii*) *Atomicity*: Enabled actions in $\mathcal{P}$ are executed one at a time, (*iii*) *Fairness*: If an action is continuously enabled along the states in the sequence, then that action is eventually chosen for execution, and (*iv*) *Maximality*: Maximality of the sequence means that if the sequence is finite then the guard of each action in $\mathcal{P}$ is false in the final state.
Closure. Let S be a state predicate. An action ac of p preserves S iff in any state where S is true and ac is enabled, atomically executing the statement of ac yields a state where S continues to be true. S is closed in a set of actions iff each action in that set preserves S.
Specification. The program specification is a set of *acceptable* computations. Following Alpern and Schneider [15], specification can be decomposed into a *safety* specification and a *liveness* specification. Also, as shown in [16], for a rich class of specifications, safety specification can be represented as a set of bad transitions that should not occur in program computations. Hence, we represent this safety specification by a set of bad transitions that should not occur in program computations. We omit the representation of liveness specification here as it is not required for our purpose.
Satisfies. $\mathcal{P}$ satisfies specification from S iff each computation of $\mathcal{P}$ that starts from a state where S is true is in the specification.

Invariant. The state predicate S of $\mathcal{P}$ is an invariant iff (i) S is closed in $\mathcal{P}$, and (ii) $\mathcal{P}$ satisfies specification from S.
Specification during adaptation. We argue that while the specification before and after adaptation can be arbitrary, 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.
Notation. We use the notation $X \veebar Y \veebar Z$ to imply that only one of X, Y, or Z is true at a time.

3 Verifying Adaptation

In this section, we recall the relevant definitions and results from [7]. These definitions include *adaptation lattice*, *transitional-invariants* and *transitional-invariant lattice*.
Adaptation lattice. An *adaptation lattice* is a finite directed acyclic graph in which each node is labeled with one or more predicates and each edge is labeled with an atomic adaptation, such that,

1. There is a single *entry node* P having no incoming edges. The entry node is associated with predicates over the variables of the *old* program, i.e., the program before adaptation. The entry node is also called a start node.
2. There is a single *exit node* Q having no outgoing edges. The exit node is associated with predicates over the variables of the *new* program, i.e., the program after adaptation. The exit node is also called an end node.
3. Each intermediate node R that has at least one incoming and outgoing edge is associated with predicates over the variables of *both* the old and the new program.

In the context of adaptation, the program adds and removes components, and thus, the program during adaptation consists of actions of the old program and 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. A *transitional-invariant* is a predicate that is true throughout the execution of an intermediate program and is closed under the old program actions that are not yet removed and the new program actions that are already added. However, the atomic adaptations 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 (see Fig. 1 for an example):

1. The entry node P is associated with an invariant S_P of the program before adaptation.
2. The exit node Q is associated with an invariant S_Q of the program after adaptation.

3. Each intermediate node R is associated with a transitional-invariant TS_R, such that any intermediate program at R (i.e., intermediate program obtained by performing adaptations from the entry node to R) satisfies the (safety) specification during adaptation from TS_R.
4. If a node labeled R_i has an outgoing edge labeled A to a node labeled R_j, then performing atomic adaptation A in any state where TS_{R_i} holds and guard of A is true results in a state where TS_{R_j} holds, and the transition obtained by A satisfies the safety specification during adaptation.
5. 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 any execution of any intermediate program at R that starts in a state where TS_R is true, eventually, the guard of at least one atomic adaptation a_i, $1 \leq i \leq k$, becomes true and remains true thereafter and, hence, eventually some a_i will be performed.

Remark. An intermediate node R could be reached by multiple paths. Therefore, it is required that TS_R be met for each intermediate program corresponding to the path.
Now, to prove the correctness of adaptation, we need to find a transitional-invariant lattice corresponding to the adaptation. This is stated formally by the following theorem: (We refer readers to [17] for the proof.)

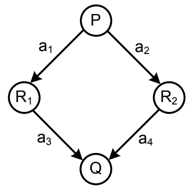
$$\begin{tikzcd}
 & P \arrow[dl, "a_1"'] \arrow[dr, "a_2"] & \\
R_1 \arrow[dr, "a_3"'] & & R_2 \arrow[dl, "a_4"] \\
 & Q &
\end{tikzcd}$$

Fig. 1. An example of a transitional-invariant lattice

Theorem. *Given S_P as the invariant of the program before adaptation and S_Q as the invariant of the program after adaptation, if there is a transitional-invariant lattice for an adaptation with entry node associated with S_P and exit node associated with S_Q, then the adaptation depicted by that lattice is correct (i.e., while the adaptation is being performed the specification during adaptation is satisfied and after the adaptation completes, the application satisfies the new specification).* □

4 Concurrency v/s Complexity of Verification

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. As a result, in the adaptation lattice, we have multiple paths from start node to end node to encompass all possible orderings among concurrent atomic adaptations. Each path from start node to end node in the lattice denotes a possible adaptation from an old program to a new program. We call a path in the lattice from the start node to the end node as an *adaptation path*.

Putting concurrent atomic adaptations in various possible orderings is a potential cause of the explosion in size of the adaptation lattice. For example, if an adaptation consists of n atomic adaptations that can be executed concurrently, then there are $n!$ different orderings and $2^n - 2$ different intermediate programs. Thus, $2^n - 2$ transitional-invariants need to be identified corresponding to each intermediate program. The adaptation lattice in this case is as shown in Fig. 2(a) for $n = 3$. To identify all these transitional-invariants and verify the corresponding intermediate programs is a difficult process.

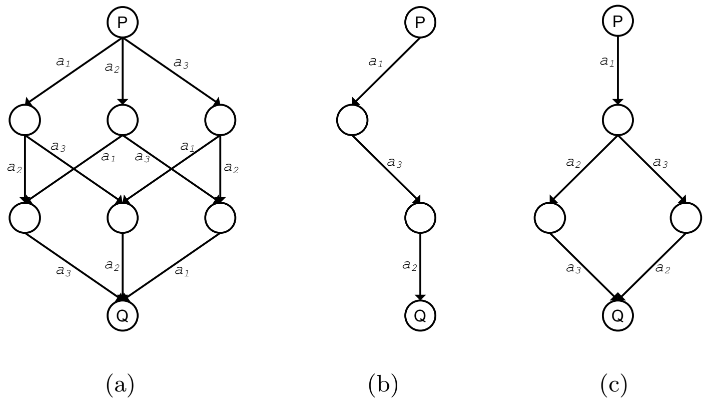

Fig. 2. Executing three atomic adaptations

Clearly, the specification during adaptation is satisfied, if the adaptation follows any path in the lattice. So instead of choosing to verify all adaptation paths, if we verify only one path (e.g. a_1, a_3, a_2), then the lattice would be as shown in Fig. 2(b). For specification during adaptation to be satisfied the adaptation must follow this path, i.e., a_1 has to occur before a_3, and a_3 has to occur before a_2. In this case there is no concurrency of adaptation, and we are able to reduce the cost of verification from $O(2^n)$ to $O(n)$. Specifically, for n concurrent atomic adaptations, the number of transitional-invariants that needs to be identified is reduced to $n - 1$.

Alternatively, we could have chosen a lattice as shown in Fig. 2(c). In this case the cost of verification is more compared to the lattice in Fig. 2(b), but less compared to the lattice in Fig. 2(a). Also, the concurrency in the lattice in Fig. 2(c) is more compared to the lattice in Fig. 2(b), but less when compared to the lattice in Fig. 2(a).

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

5 Case Study

In this section, we illustrate the tradeoff between concurrency of adaptation and complexity of verifying that adaptation using the replacement of mutual exclusion protocols. We describe adaptation from a centralized mutual exclusion protocol to a distributed mutual exclusion protocol, and demonstrate how the ordering among atomic adaptations reduces the size of the adaptation lattice. We choose these two extreme solutions of mutual exclusion protocols, because it is easy to explain the tradeoff in this context.

We use the *component family* design [18] to model these two protocols. The component family design focuses on building a family of components involved in providing similar functionality. For separating the adaptation concern, each component in a family consists of two parts (*i*) a *functional part*, and (*ii*) an *adapt-active* part. The adapt-active part consists of actions that are needed only during adaptation.

In this case study, we build a family of mutual exclusion protocols. We consider two protocols in the family, a centralized mutual exclusion protocol and a distributed mutual exclusion protocol. We consider mutual exclusion by Felten and Rabinovich [8] for centralized mutual exclusion protocol, and by Ricart and Agrawala [9] for distributed mutual exclusion protocol.

In the following subsections, we describe the details of these protocols. These protocols are installed on a distributed program consisting of N processes. Each process is denoted by i ($1 \leq i \leq N$). Each process i has two output variables try_i and $reset_i$, which are inputs to the mutual exclusion protocol; and two input variables $crit_i$ and rem_i, which are outputs of the mutual exclusion protocol. When a process wants access to the critical section, it sets try_i to true. The process can access the critical section only when $crit_i$ is set to true. When it has finished accessing the critical section it sets $reset_i$ to true. When a process is not accessing or requesting the critical section, the mutual exclusion protocol sets rem_i to true.

5.1 Centralized Mutual Exclusion Protocol

In centralized mutual exclusion protocol, one of the node acts as a master node. The role of the master node is to assign the permission to the critical section to the requesting nodes. If a client needs to access the critical section, it requests the token from the master. The master maintains a list of nodes that have requested the token. It grants the token to the first node in the queue. A client node after finishing its critical section informs the master; whereupon the master will remove that client from the list, and grant the token to the next node in the list.

The centralized mutual exclusion protocol is shown in Fig. 3. Each fraction CentralME.f_i of the protocol is installed at the corresponding process i of the distributed program. The master fraction is denoted by CentralME.m ($m = f_i$ for some i). A client requests the token by sending `REQUEST` message to the master. The master maintains two data structures: the *activeQ* contains a list of nodes that have requested the token, and which will receive the token without further action by the master; the *passiveQ* is a list of nodes that have requested the token, but whose requests will not be satisfied until the master sends further messages. A client node receives the token by the arrival of the `GRANT` message. The `GRANT` message contains a queue of nodes that have requested the token. When a client

```
**Component** CentralME
const $minLen, m$
Fraction CentralME.$f_i$
  input   $try_i, reset_i$            : bool
  output  $crit_i$                    : bool   {initially false}
          $rem_i$                     : bool   {initially true}
  var     $afterMe, forwardQ$         : list   {initially $\phi$}
          $lastbatch$                 : int    {initially 0}
          $critReq, sendingReq$       : bool   {initially false}
          $activeQ, passiveQ$         : list   {initially $\phi$}
          $batch$                     : int    {initially 0}

  rcvReq    : $f_i = m \wedge$ rcv REQUEST from $f_j \ \rightarrow\ passiveQ := passiveQ \circ j$;
  rcvDone   : $f_i = m \wedge$ rcv DONE from $f_j \ \rightarrow\ activeQ :=$ removeMatchedItem($activeQ, j$);
  Grant     : $f_i = m \ \wedge activeQ = \phi \wedge passiveQ \neq \phi \rightarrow\ activeQ := passiveQ$;
                                                $passiveQ := \phi$;
                                                send GRANT($batch, activeQ$) to head($activeQ$);
                                                $batch := batch + 1$;
  Forward   : $f_i = m \wedge$ length($activeQ$) $< minLen \ \wedge\ passiveQ \neq \phi \rightarrow$
                                                send FORWARD($batch, passiveQ$) to tail($activeQ$);
                                                $batch := batch + 1$;
                                                $activeQ := activeQ \circ passiveQ$;
                                                $passiveQ := \phi$;
  start_try : $try_i \wedge \neg critReq \rightarrow\ critReq, sendingReq :=$ true;
                                   $rem_i :=$ false;
  sndReq    : $critReq \wedge sendingReq \ \rightarrow$ send REQUEST to $m$;
                                   $sendingReq :=$ false;
  allowCrit : rcv GRANT($bnum, Q$) from $m \vee$ rcv GRANT($bnum, Q$) from $f_j \rightarrow$
                                                $crit_i :=$ true;
                                                $critReq :=$ false;
                                                $lastbatch := bnum$;
                                                $afterMe :=$removeFirstItem($Q$);
  exit      : $crit_i \wedge reset_i \rightarrow\ crit_i :=$ false;
                           $rem_i :=$ true;
                           send DONE to $m$;
                           if $forwardQ \neq \phi$
                              $afterMe := afterMe \circ forwardQ$;
                              $forwardQ := \phi$;
                              $lastbatch := lastbatch + 1$;
                           if $afterMe \neq \phi$
                              send GRANT($lastbatch, afterMe$) to head($afterMe$);
  rcvFwd    : rcv FORWARD($bnum, Q$) from $m \rightarrow$ if $bnum = lastbatch + 1 \wedge \neg crit_i$
                                                   send GRANT($bnum, Q$) to head($Q$);
                                                elseif $bnum > lastbatch$
                                                   $forwardQ := Q$;
```

Fig. 3. Centralized Mutual Exclusion Protocol [8]

receives a GRANT message, it removes its own ID from the head of the queue, and after finishing its critical section, it sends the GRANT message to the first node in the queue. A client node after finishing its critical section also sends a DONE message to the master. A master removes the ID of the node from its *activeQ* after it receives DONE from that node.

The invariant of the centralized mutual exclusion protocol is as shown in Fig. 4. Informally, the invariant states that the execution of CentralME protocol satisfies the following conditions: (*i*) if a process is in the master's *activeQ* list, then either it is accessing the critical section or its request is still not granted, (*ii*) if a process is in the master's *passiveQ* list, then its request to access the critical section is still not granted, and (*iii*) if a process is accessing the critical section, then all processes who requested critical section before it have finished accessing the critical section,

$T_1 \wedge T_2 \wedge T_3$, where

$$T_1 = \forall i : f_i \in m.activeQ \Rightarrow f_i.critReq \veebar crit_i$$

$$T_2 = \forall i : f_i \in m.passiveQ \Rightarrow f_i.critReq \wedge \neg crit_i$$

$$\begin{aligned} T_3 = \forall i : crit_i \Rightarrow {} & f_i \in m.activeQ \wedge \\ & \forall j : f_j \in \texttt{precedeList}(m.activeQ, f_i) :: \#ch.f_j.m(\texttt{DONE}) = 1 \wedge \\ & \forall k : f_k \in \texttt{succeedList}(m.passiveQ, f_k) :: (f_k.critReq \wedge \neg crit_k) \end{aligned}$$

Fig. 4. Invariant of CentralME

and all processes who requested critical section after it are still waiting for access to critical section.

5.2 Distributed Mutual Exclusion Protocol

In distributed mutual exclusion protocol, the nodes communicate with each other to arrive at a consensus on what process will get access to the critical section. There are several distributed mutual exclusion protocols; most of them can be classified into two categories: quorum-based and token-based protocols. In quorum-based protocols, each node maintains its own local lock that can be given to one process at a time. In order to attain a mutual exclusion, a process must obtain local locks on a set of nodes called a *quorum*. Quorums are defined in such a way that any two of them have at least one node in common, which ensures at most one process can succeed in obtaining local locks from a quorum. In token-based protocols, the mutual exclusion privilege is associated with the possession of a *token*. The protocols ensure that there is always only one valid token in the system, and hence, mutual exclusion is enforced. We consider the Ricart-Agrawala protocol [9] for achieving mutual exclusion in a network, which is one of the well-known protocols in distributed computing.

The distributed mutual exclusion protocol is shown in Fig. 5. A node making an attempt to invoke mutual exclusion sends a `REQUEST` message to all other nodes. Upon receipt of the `REQUEST` message, the other node either sends a `REPLY` immediately or defers a response until after it leaves its own critical section. A `REPLY` message is returned immediately if the originator of the `REQUEST` message has priority; otherwise, the `REPLY` is delayed. The priority order decision is made by comparing a sequence number present in each `REQUEST` message. If the sequence numbers are equal, the node numbers are compared to determine which node will get access to critical section first. A node can access the critical section after all other nodes have sent their `REPLY` messages.

The invariant of the distributed mutual exclusion protocol is as shown in Fig. 6. Informally, the invariant states that the execution of RicartAgrawalaME satisfies the following conditions: (i) if f_j has received a request from f_i, then the highest sequence number (*highestseqno*) of f_j is greater than or equal to the sequence number (*seqno*) of f_i, and (ii) if f_j has received a request from f_i, it will grant its permission to f_i if f_j itself is not requesting for access to the critical section, or the sequence number of f_i is less than f_j's sequence number, or if the sequence numbers are same, then $i < j$.

Notation. Instead of CentralME.f_i (respectively, RicartAgrawalaME.f_i), we use f_i if the component CentralME (respectively, RicartAgrawalaME) is obvious from

```
Component RicartAgrawalaME
const $N$
Fraction RicartAgrawalaME.$f_i$
  input  $try_i, reset_i$           : bool
  output $crit_i$                   : bool   {initially false}
         $rem_i$                    : bool   {initially true}
  var    $seqno$                    : int    {initially arbitrary}
         $highestseqno$             : int    {initially 0}
         $replycnt$                 : int    {initially arbitrary}
         $critReq, sendingReq$      : bool   {initially false}
         $defReply$                 : array [1 : $N$] of bool
                                             {initially false}

  start_try  : $try_i \wedge \neg critReq \;\rightarrow\; critReq, sendingReq :=$ true;
                                   $rem_i :=$ false;
  sndReq     : $critReq \wedge sendingReq \;\rightarrow\; seqno := highestseqno + 1$;
                                   $replycnt := N - 1$;
                                   for $p = 1$ to $N$
                                      if $i \neq p$
                                         send REQUEST(seqno) to $f_p$;
                                   $sendingReq :=$ false;
  rcvRep     : rcv REPLY from $f_j \;\rightarrow\; replycnt := replycnt - 1$;
  allowCrit  : $critReq \wedge replycnt = 0 \;\rightarrow\; crit_i :=$ true;
                                   $critReq :=$ false;
  exit       : $crit_i \wedge reset_i \;\rightarrow\; crit_i :=$ false;
                                   $rem_i :=$ true;
                                   for $j = 1$ to $N$
                                      if $defReply[j]$
                                         $defReply[j] :=$ false;
                                         send REPLY to $f_j$;
  rcvReqt    : rcv REQUEST(k) from $f_j \;\rightarrow$ if $highestseqno < k$
                                      $highestseqno := k$;
                                   $defReply[j] := critReq \;\wedge\; (seqno, i) \prec (k, j)$;
                                   if $\neg defReply[j]$
                                      send REPLY to $f_j$;
```

Fig. 5. RicartAgrawala Mutual Exclusion Protocol [9]

$S_1 \wedge S_2$, where

$S_1 = \text{request\_received}(f_i, f_j) \Rightarrow f_i.seqno \leq f_j.hsn$

$S_2 = \text{granted}(f_i, f_j) \Rightarrow \neg f_j.critReq \;\vee\; (f_i.seqno, i) \prec (f_j.seqno, j)$

$\text{granted}(f_i, f_j) = \text{request\_received}(f_i, f_j) \wedge \neg f_j.defRep[i]$

$\text{request\_received}(f_i, f_j) = \text{requested}(f_i, f_j) \wedge \#ch.f_i.f_j(\texttt{REQUEST}) = 0$

$\text{requested}(f_i, f_j) = f_i \neq f_j \;\wedge\; (f_i.(\texttt{rcv REPLY from } f_j) \vee$
$(f_i.\mathsf{critReq} \wedge \neg f_i.sendingReq) \;\vee\; (f_i.\mathsf{critReq} \wedge f_i.replycnt = 0))$

Fig. 6. Invariant of RicartAgrawalaME

the context. We refer the variable *var* of fraction f_i as $f_i.var$, and of process i as var_i.

5.3 Adaptation

Centralized mutual exclusion protocols requires less message passing between nodes, but is vulnerable to load on the master. When the master node is under heavy load, the system should dynamically adapt to a distributed mutual exclusion protocol. We consider the adaptation from the centralized mutual exclusion protocol (cf. Fig. 3) to the distributed mutual exclusion protocol (cf. Fig. 5).

One way to do this adaptation would be to initially stop all the processes from requesting access to critical section. After all processes that requested access to the critical section have finished accessing the critical section, the corresponding fractions can be changed. The processes can start requesting access to critical section only when the adaptation (protocol replacement) is complete. This approach is similar to doing the adaptation statically, i.e., to stop the application and recompile (or reload) the application with the new protocol and then restart the application. This approach of doing adaptation leads to unnecessary interruption and blocking.

Another way of doing adaptation is to do the protocol replacement without blocking (stopping) the entire application. It would be desirable to let processes continue requesting access to critical section while the protocols are being replaced. The adaptation is done as shown in Fig. 7, which describes the adapt-active parts of each fractions that are involved in the adaptation. The adapt-active parts of the fractions contain the atomic adaptations that are associated with them. The name of each atomic adaptation has two subscripts (e.g., a_{1i}). The first subscript denotes the order in the sequence of atomic adaptation. Since all atomic adaptations has the same first subscript, it means that they can be executed concurrently (in any order). The second subscript denotes the process that the atomic adaptation is associated with. At each process i, there are two atomic adaptations that have same name a_{1i}, one associated with CentralME.f_i fraction and other associated with RicartAgrawalaME.f_i fraction. This implies that removal of CentralME.f_i and addition of RicartAgrawalaME.f_i needs to be done in an atomic manner.

```
Adapt-active parts
  Fraction : CentralME.$f_i$
    $a_{1i}$ : $\neg crit_i \rightarrow$ remove $f_i$;
  Fraction : RicartAgrawalaME.$f_i$
    $a_{1i}$ : $\mathsf{a}_{1i}$ (CentralME.$f_i$) $\rightarrow$ add $f_i$;
```

Fig. 7. Adaptation from CentralME to RicartAgrawalaME

In this adaptation, the fraction at a process is replaced if that process is not accessing the critical section. During adaptation, some process will have the old fractions running while some will have the new fractions running. If the new (RicartAgrawalaME) fraction receives any message (`REQUEST`, `DONE`, `GRANT`) from the old (CentralME) fraction, then that message is discarded. Any message (`REQUEST`) sent by the new (RicartAgrawalaME) fraction to the process still using the old (CentralME) fraction is buffered, and is made available to the new fraction once it replaces the old fraction at that process. In this adaptation, the process with new fraction can start requesting access to critical section before the adaptation completes. However, that process will be granted access to critical section only when all processes have completed their atomic adaptation. Further, the system does not have to wait for all processes to be out of critical section before initiating adaptation.

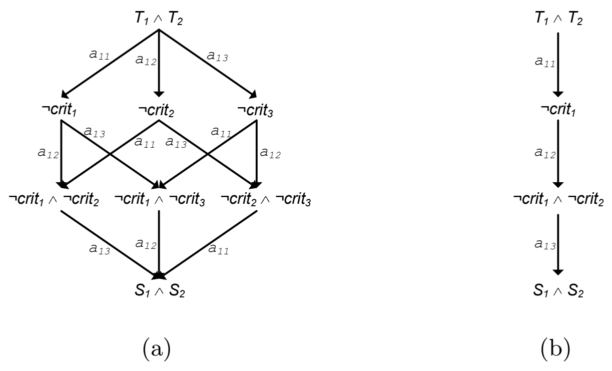

Fig. 8. Transitional-invariant lattices

The transitional-invariant lattice in this case consists of $2^N - 2$ transitional-invariants, and $N!$ orderings as atomic adaptations are interleaved in all possible ways. To verify the adaptation, each node and each edge in the lattice needs to be verified independently. The complexity of the verification grows exponentially with the number of processes. The transitional invariant lattice for the adaptation of Fig. 7 for $N = 3$ is shown in Fig. 8(a). To simplify the verification, we model the adaptation as discussed next.

Constrained Adaptation. To simplify verification, we model the adaptation by introducing constraints on the atomic adaptations. We introduce priority among the processes such that a process having a lower ID executes an atomic adaptation before a process having a higher ID. This is shown in Fig. 9. In this case, as shown in Fig. 8(b), there is only one ordering and $n - 1$ transitional-invariants.

Adapt-active parts

Fraction : CentralME.f_i

  $\boldsymbol{a_{1i}}$: $(i \neq 1 \wedge \mathsf{a}_{1(i-1)}) \wedge \neg crit_i \rightarrow$ `remove` f_i;

Fraction : RicartAgrawalaME.f_i

  $\boldsymbol{a_{1i}}$: a_{1i} (CentralME.f_i) $\rightarrow$ `add` f_i;

Fig. 9. Constrained Adaptation from CentralME to RicartAgrawalaME

6 Discussion

In this section, we explain some of the issues related to our work.

How is the verification of the program before adaptation and the program after adaptation related to our approach? Why is it necessary to verify program during adaptation?

Since the program and the specification before adaptation is fixed, we can use existing techniques, such as model-checking and theorem-proving, to verify the program before adaptation. These techniques can also be used to verify the program after adaptation, as the program and specification after adaptation is also fixed, though different from the old program and specification. However, verifying the old and new program is not enough, as the specification during adaptation is also to be satisfied. For example, as discussed in Section 5, while replacing mutual exclusion protocol we still want to satisfy mutual exclusion property during adaptation. Moreover, since the program during adaptation is not fixed, it is not straightforward to use the existing techniques such as model-checking and theorem-proving for verification. As shown in the Section 3, the intermediate programs and the transitional-invariants need to identified for verifying program during adaptation. The results of verification of the old and the new program are used in instantiating the start node and the end node in the lattice.

How does enforcing ordering among atomic adaptations affect the execution of adaptation?

Executing the atomic adaptations in a predefined order may increase the amount of time it takes to complete the adaptation compared to the case where these atomic adaptations can be executed concurrently. However, since all atomic adaptations need to be executed in order for the adaptation to complete, a delay in any atomic adaptation may delay the entire adaptation. In order to reduce the verification complexity while minimizing the execution time of adaptation, one can enforce the ordering only on a subset of concurrent atomic adaptations.

Can the verification complexity be reduced in presence of concurrent atomic adaptations?

In this paper, we reduced the verification complexity by introducing constraints on the adaptation to enforce some ordering among concurrent atomic adaptations. In case where the concurrent atomic adaptations are similar, the different adaptation paths in the adaptation lattice would be also similar. If it is possible to verify a subset of these paths and imply the correctness of other paths based on the similarity among atomic adaptations, then this approach will reduce the verification complexity in presence of concurrent atomic adaptations.

7 Conclusion

In this paper we discussed an approach based on the adaptation lattice to verify the dynamic adaptation and the tradeoff between concurrency of adaptation and complexity of verifying that adaptation. We discussed how the enforcing of ordering among atomic adaptations can reduce the complexity of verifying adaptation. We presented our results using the case study of dynamic adaptation from centralized mutual exclusion protocol to a distributed mutual exclusion protocol.

The results from our case study in this paper suggests that by setting appropriate constraints on adaptation, we can reduce the complexity of verifying the adaptation, and thereby, making it easy to provide assurance guarantees for adaptation.

As discussed in Section 6, we would like to reduce the verification complexity without altering the concurrency of adaptation. In this context, we would like to investigate if it is possible to provide assurance of the entire lattice by verifying only a subset of the lattice. Though not formally established, our current results suggest that this may be possible in case where there is similarity among adaptation paths in the lattice. As a part of future work, we want to use some of the results from partial order reduction in model-checking and apply them in the context of verifying adaptation using adaptation lattice.

References

1. 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.
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. 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.
4. B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behavior. In *ECOOP*, pages 205–230, 2002.
5. R. Keller and U. Hölzle. Binary component adaptation. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1998.
6. S. Masoud Sadjadi. *Transparent Shaping of Existing Software to Support Pervasive and Autonomic Computing*. PhD thesis, Michigan State University, 2004.
7. Sandeep Kulkarni and Karun Biyani. Correctness of component-based adaptation. In *International Symposium on Component-based Software Engineering - CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 48–58, May 2004.
8. E. W. Felten and M. Rabinovich. A centralized token-based algorithm for distributed mutual exclusion. Technical Report 92-02-02, Department of Computer Science and Engineering, University of Washington, February 1992.
9. G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.
10. P. McKinley and U. Padmanabhan. Design of composable proxy filters for mobile computing. In *Workshop on Wireless Networks and Mobile Computing*, 2001.
11. M. Gouda. *Elements of Network Protocol Desgin*. John Wiley & Sons, 1998.
12. Gerard Holzmann. Logic verification of ansi-c code with spin. *The Sixth SPIN Workshop*, 2000.
13. M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.
14. M. Demirbas and A. Arora. Convergence refinement. *International Conference on Distributed Computing Systems*, 2002.
15. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985.
16. A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE transactions on Software Engineering*, 24(1):63–78, January 1998.
17. 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.
18. Karun Biyani and Sandeep Kulkarni. Building component families to support adaptation. In *International Workshop on Design and Evolution of Autonomic Application Software - DEAS, at ICSE*, May 2005.