Concurrency Tradeoffs in Dynamic Adaptation[1]

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

Abstract

Software systems need to adapt as requirements change, environment conditions vary, and bugs are discovered and fixed. 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. In our previous work, we presented an approach based on transitional-invariant lattice for verifying the correctness of dynamic adaptation. The size (number of nodes and edges) of the lattice affects the complexity of verifying and testing the adaptation. The size of the lattice itself is dependent on the number of steps in the adaptation, and the concurrency (dependency) among those steps. In this paper, we discuss various tradeoffs that arise because of concurrency during adaptation.

Key words: Dynamic Adaptation, Specification, Verification, Concurrency, Complexity

1 Introduction

Many computing systems are required to provide uninterrupted service. These systems need to be updated as bugs are discovered, and need to be adapted as environment changes. If these systems are to provide continuous operation, the update and the adaptation needs to be done without completely stopping the system. Such adaptation is commonly referred to as dynamic adaptation.

Dynamic Adaptation can be classified based on different parameters, such as: (1) distributed *vs* single process, (2) code change *vs* parameter change, (3) requiring state-transfer *vs* no state-transfer, (4) application-specific *vs* general-purpose, (5) anticipated *vs* unanticipated, (6) language-dependent *vs* language-independent. The kind of adaptation that we consider is classified as compositional adaptation, one that modifies the system by adding, removing, or replacing components (code) in distributed systems. We consider this type of adaptation as it is more general form of adaptation, and results derived in this context can be applied to other form of adaptation such as single-process, or parameter-based 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, verifying the system during adaptation and the system after adaptation. Approaches in [1–6] have focused on syntactic issues in adaptation in distributed systems. Semantic issues have been considered in [7–10]. However, these approaches are limited in that the approaches in [7–9] focus on offline adaptation, whereas the approach in [10] focuses on online adaptation of a single process system (that can also be extended to distributed systems that communicate via RPC). However, the issue of correctness of a distributed program before, during, and after adaptation is not adequately addressed in the literature.

To redress this deficiency, in [11], we focused on an approach for verifying adaptation in a distributed system. Compositional adaptation in distributed systems involves modification to several (if not all) processes involved in the system. For example, an adaptation that replaces a network protocol X with a network protocol Y in a system, will require replacing each *protocol fraction* of X with corresponding *protocol fraction* of Y at every process. Clearly, such adaptation cannot be performed atomically, and is typically considered as a multi-step procedure. We, therefore, introduced the notion of *atomic adaptation*; typically only a single process will be involved in an atomic adaptation. An atomic adaptation may involve blocking or replacing a fraction. Thus, the adaptation in a distributed system consists of several atomic adaptations. It follows that during adaptation, the system may consist of parts of the old program (i.e., program before adaptation that is using old protocol fractions) and parts of the new program (i.e., program after adaptation that is using new protocol fractions). Hence, to guarantee correctness of adaptation, in [11], we focus on the properties of such *intermediate programs*.

To verify the properties of a system during adaptation

[1] 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.

and system after adaptation, in [11], we introduced an *transitional-invariant lattice*. Specifically this lattice enables us to verify properties of system during 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. Further, the size of the lattice is dependent on the *dependency* ([1]) among the protocol fractions. Based on the *dependency relation* among protocol fractions, the atomic adaptations are either performed independently (concurrently) or in a specific sequence.

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 during 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. Further, we show that in case of adaptation, reducing concurrency among atomic adaptations not only reduces the verification complexity, but may also reduce the communication overhead during adaptation.

Organization of the paper. In Section 2, we discuss overview of the adaptation model. We describe the transitional-invariant lattice used in verification of adaptation in Section 3. The concurrency tradeoffs during adaptation are discussed in Section 4. We present case studies in Section 5. In Section 6, we discuss some issues related to our work, and finally conclude in Section 7.

2 Modeling Adaptation

We consider adaptations where the program adds, removes, or replaces a network (or distributed) protocol. A distributed 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 (encrypted) communication between a sender and a receiver. Such a protocol consists of two fractions, namely, an *encryption fraction* at the sender that encrypts the packets before sending and a *decryption fraction* at the receiver that decrypts the encrypted packets received from the sender. Other examples of distributed protocol include components that implement group communication protocols such as leader election and mutual exclusion, and routing protocols. For discussion in this section, we consider the following general example of adaptation: consider a program consisting of n processes, and let X and Y be two distributed protocols. Both protocols X and Y have a fraction that is installed at every process. The program before adaptation is using the protocol X. The adaptation in the system replaces the protocol X with the protocol Y, resulting in the new program that uses the protocol Y. Typically, such a compositional adaptation in distributed systems consists of three phases: (i) initialization phase, (ii) synchronization phase, and (ii) completion phase.

- **Initialization phase**. Usually, the adaptation in distributed system involves an initialization phase in which the processes decide and prepare for adaptation. The decision about when to start adaptation is either taken by a single process, or a group of processes, or an external entity. Regardless of who makes this decision, or how this decision is made, the decision needs to be communicated to all processes involved in the adaptation. At the end of initialization phase, all processes are ready to start the adaptation, and enter the next phase called *synchronization phase*.
- **Synchronization phase**. The goal of the synchronization phase is to replace a protocol X with protocol Y. To replace X with Y, each fraction of X needs to be replaced with each fraction of Y. As discussed in introduction and in more detail in [1], there may exists *dependency* among fractions, because of which the fractions cannot be replaced independently. The replacement of fractions need to be done in a synchronized way to handle any dependencies that may exist among fractions. Thus, an adaptation in a distributed program involves multiple steps. The key to verifying adaptation in distributed programs is ensuring that the individual atomic steps in adaptation occur in an appropriate order (to handle dependency) and the instances when they occur are "safe", i.e. the *specification during adaptation* is satisfied. Towards this end, we divide this multi-step adaptation into multiple *atomic adaptations* each occurring at only one process. Based on the dependency among protocol fractions, some (or all) of these atomic adaptations may occur concurrently. Concurrent atomic adaptations means that the atomic adaptations are independent of each other and execution of one does not affect execution of others. As is the case with verifying concurrent systems, to verify adaptation we consider all possible interleavings among concurrent atomic adaptations.
- **Completion phase**. In the completion phase, all processes are informed about the end of synchronization phase. This phase may be optional depending on the type of adaptation and requirements. At the end of completion phase, each process knows that all processes in the system have finished adaptation.

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 another intermediate program. The

last atomic adaptation results into the new program. Identifying intermediate programs is important for verification (and testing), as it is hard (if not impossible) to argue about correctness of a changing system by considering it as a single program.

3 Verifying Adaptation

In this section, we recall some of the relevant definitions and results from [11]. We refer the reader to [11] for formal definitions of other terms such as invariant, closure, and specification.

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.

Transitional-invariant lattice. A *transitional-invariant lattice* is a finite directed acyclic graph with each node having one predicate and that satisfies the following five conditions (cf., Fig. 1 for an example):

1. There is a single *entry node* P having no incoming edges. The entry node is associated with an invariant S_P of the program before adaptation.
2. There is a single *exit node* Q having no outgoing edges. The exit node is associated with an invariant S_Q of the program after adaptation.
3. Each intermediate node R has at least one incoming edge and at least one outgoing edge. It 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.

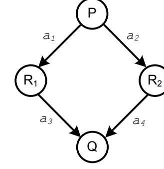


Figure 1. An example of a transitional-invariant lattice.

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 [12] for the proof.)

Theorem 1. *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 during Adaptation

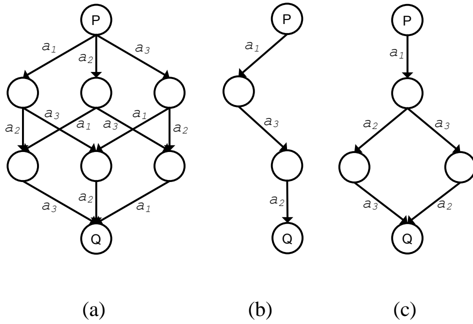
Concurrent executions are generally considered faster than sequential executions. Specifically, with respect to adaptation, we would expect that concurrent execution of atomic adaptations (if possible after considering any dependencies) would be faster than sequential execution. However, time of adaptation is not the only factor that needs to be considered while designing adaptation. As we discuss in the following subsections, verification complexity increases exponentially with increase in concurrency, and also message overhead increases with increase in concurrency.

4.1 Concurrency v/s Verification Complexity

As discussed in Section 2, to verify a given adaptation, we need to consider all possible orderings of concurrent atomic adaptations. As a result, in the lattice, we have multiple paths from start node to end node to encompass all possible orderings among concurrent atomic adaptations. In other words, this increases the number of intermediate programs that need to be verified.

Putting concurrent atomic adaptations in various possible orderings is a potential cause of the explosion in size of the 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 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.



(a) (b) (c)

Figure 2. Executing three atomic adaptations.

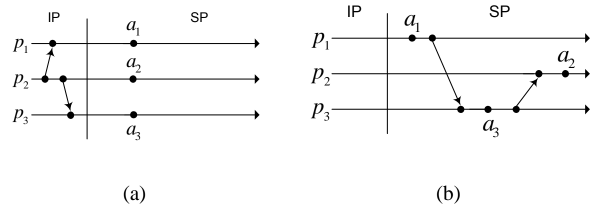
Clearly, the specification during adaptation is satisfied, if the adaptation follows any path in the lattice. So instead of choosing to verify all adaptation paths, if we verify only one path (e.g. a_1, a_3, a_2), then the lattice would be as shown in Fig. 2(b). For specification during adaptation to 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 during 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 lattice as defined in Section 3. The adaptation in this case has to be constrained so that it follows some path in the sublattice.

4.2 Concurrency v/s Time and Message Complexity

There are systems that are affected by communication overhead and message delays. Specifically, for wireless and mobile systems, energy-communication tradeoff may require system to reduce communication overhead whenever possible . For designing adaptation in such systems, communication overhead should also be taken into account. In this subsection, we show how concurrency during adaptation affects the communication overhead.

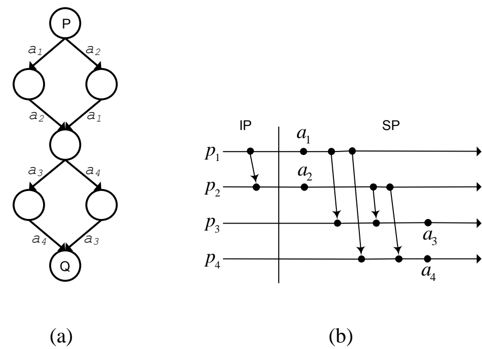


(a) (b)

Figure 3. Space-time diagram of adaptation.

Consider a case where all atomic adaptations are executed concurrently. This is described by the lattice of Fig. 2(a), and the space-time diagram for this adaptation is as shown in Fig. 3(a). We show only the minimum number of adaptation-specific messages in the space-time diagram. There may be other application-specific messages that we do not consider as they are not related to adaptation. In Fig. 3(a), process p_2 is the initiator that informs other processes to start performing any steps required for adaptation. In this case, during initialization phase there are at least two messages.

Now, if adaptation were to occur according to the lattice of Fig. 2(b), then we can make process p_1 as the initiator, and the space-time diagram would be as shown in Fig. 3(b). In this case, we got rid of initialization messages that were required for adaptation described by Fig. 3(a). In both the cases, the minimum number of adaptation-specific messages required during adaptation is two. Thus, we did not increase any communication overhead by reducing concurrency. We now consider another scenario where the number of messages can actually be reduced if concurrency is reduced during adaptation.



(a) (b)

Figure 4. Adaptation with concurrency.

Consider the lattice of Fig. 4(a) that describes adaptation consisting of four atomic adaptations a_1, a_2, a_3, and a_4 occurring at processes p_1, p_2, p_3, and p_4 respectively. Atomic adaptations a_1 and a_2 are independent of each other and can occur concurrently. Similarly, a_3 and a_4 can occur concurrently. The corresponding space-time diagram is as shown in Fig. 4(b). The minimum number of adaptation-specific messages required during adaptation is five. Now, if were to reduce concurrency in this case, we can have an adaptation

that is described by the lattice of Fig. 5(a), whose corresponding space-time diagram is as shown in Fig. 5(b). In this case, the minimum number of adaptation-specific messages required is reduced to four.

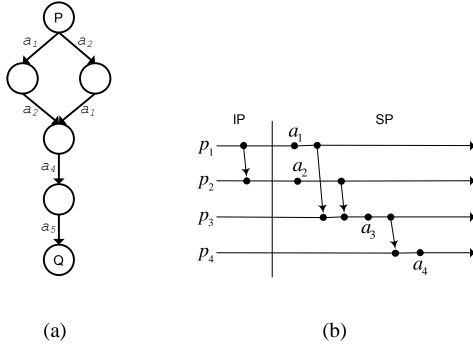


Figure 5. Adaptation with (reduced) concurrency.

Further, if we have no concurrency during adaptation as described by the lattice of Fig. 6(a), then the space-time diagram would be as shown in Fig. 6(b). In this case, a minimum of only three adaptation-specific messages are required during adaptation.

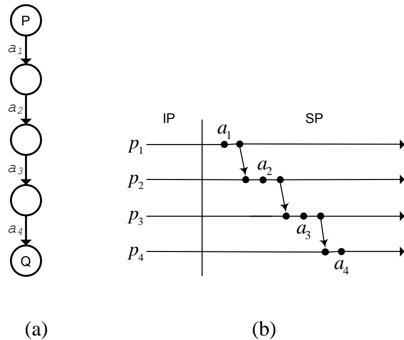


Figure 6. Adaptation with no concurrency.

Thus, by reducing concurrency during adaptation, it is possible to reduce the number of messages required during adaptation. However, from the space-time diagrams of Fig. 3-6, it is clear that time required to complete adaptation would probably be less when there is more concurrency during adaptation. Thus, while designing adaptation, one should consider various factors such as concurrency during adaptation, message delays and communication overhead, and verification complexity.

5 Case Study

In this section, we illustrate the tradeoffs due to concurrency during adaptation using the replacement of mutual exclusion and reliable communication protocols. Due to limitation of space, we only discuss overview of this case studies here, and refer reader to [11, 13] where these examples are discussed in detail.

5.1 Replacing 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 lattice. We choose these two extreme solutions of mutual exclusion protocols, because it is easy to explain the tradeoff in this context. Centralized mutual exclusion protocols requires less message passing between nodes, but performs poorly under heavy load due to the master being a bottleneck. When the master node is under heavy load, the system should dynamically adapt to a distributed mutual exclusion protocol.

We perform this adaptation of protocol replacement without blocking (stopping) the entire application. The processes continue requesting access to critical section while the protocols are being replaced. 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 fraction receives any message from the old fraction, then that message is discarded. Any message sent by the new fraction to the process still using the old 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 the 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 adaptations. Further, the system does not have to wait for all processes to be out of critical section before initiating adaptation.

In this adaptation, all atomic adaptations are independent of each other and can be performed concurrently as long as their local guards are true. The transitional-invariant lattice in this case (for three processes) is similar to one shown in Fig. 2(a). As discussed in Section 4, we reduce verification complexity and communication overhead by constraining the adaptation such that replacement of fractions is done in a sequential manner.

5.2 Replacing Reliable Communication Protocols

We consider a simple publish-subscribe system having two publishers (senders) and two subscribers (receivers). Both receivers subscribe to receive data from both the senders. For reliable communication we consider two protocols: a *proactive protocol* based on forward error correction, and a *reactive protocol* based on acknowledgments.

In this case study, the adaptation replaces proactive protocol in the system with the reactive protocol. The adaptation is done by first replacing the two senders. The atomic adaptations of replacing protocol fractions at senders can be done concurrently. We note that the local guards of these

atomic adaptations need to be true before they can be executed. Once the protocol fractions at both the senders are replaced, the protocol fractions at two receivers can be replaced (once corresponding local guards are true). These two atomic adaptations can also occur concurrently. The transitional-invariant lattice in this case is as shown in Fig. 4(a). As discussed in Section 4, the verification complexity and communication overhead can be reduced by reducing concurrency (cf. Fig. 5-6).

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 are 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 are 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. We may also want to ensure that system does not deadlock during adaptation or any other property is not violated while the system is adapting. 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 be identified for verifying program during adaptation. The results of verification of the old and the new program are used in instantiating the start and end node in the lattice.

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

If such a scenario is desired then we can design a *default component* which is equivalent to having no component at all. For example, in the context of our case study in Section 5, the default component provides no mutual exclusion. Thus, removal of a component is equivalent to replacing that component by a default component. This approach is similar to that in [1].

7 Conclusion

In this paper we discussed an approach based on the transitional-invariant lattice to verify the dynamic adaptation and various tradeoffs due to concurrency during adaptation. We discussed how the enforcing of ordering among atomic adaptations can reduce the complexity of verifying adaptation. We also showed how communication overhead during adaptation can be reduced by reducing concurrency during adaptation.

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. Further, reducing concurrency during adaptation can be beneficial for systems where message delays and communication overhead are significant. Thus, while designing adaptation, time of execution, communication overhead and verification complexity are various tradeoffs that need to be considered.

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] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] Karun Biyani and Sandeep Kulkarni. Concurrency and complexity in verifying dynamic adaptation: A case study. Technical Report MSU-CSE-05-21, Michigan State University, July 2005.