Efficient Dynamic Updates of Distributed Components Through Version Consistency

Luciano Baresi, Carlo Ghezzi, *Fellow, IEEE*, Xiaoxing Ma, *Member, IEEE*, and Valerio Panzica La Manna

Abstract—Modern component-based distributed software systems are increasingly required to offer non-stop service and thus their updates must be carried out at runtime. Different authors have already proposed solutions for the safe management of dynamic updates. Our contribution aims at improving their efficiency without compromising safety. We propose a new criterion, called *version consistency*, which defines when a dynamic update can be safely and efficiently applied to the components that execute distributed transactions. Version consistency ensures that distributed transactions be served as if they were operated on a single coherent version of the system despite possible concurrent updates. The paper presents a distributed algorithm for checking version consistency efficiently, formalizes the proposed approach by means of a graph transformation system, and verifies its correctness through model checking. The paper also presents ConUp, a novel prototype framework that supports the approach and offers a viable, concrete solution for the use of version consistency. Both the approach and ConUp are evaluated on a significant third-party application. Obtained results witness the benefits of the proposed solution with respect to both timeliness and disruption.

Index Terms—Component-based distributed system, dynamic update, version-consistency

1 Introduction

Many modern software systems are increasingly required to offer continuous, *non-stop* services. For example, large information systems in the medical domain, financial transaction processing applications, and critical systems that control flights and trains must be continuously available and cannot be shut down. Traditional software maintenance supports software evolution by providing updates that are applied *off-line*. That is, the system is shut down, updated, and restarted. This solution, however, is not applicable when the system must always be on. *Dynamic updates*, that is, changes that are applied while the system is in operation, become mandatory.

Compared to off-line maintenance, *safe* dynamic updates are more difficult: in addition to the correctness of the new version, they must also preserve the correct completion of all on-going activities. At the same time, dynamic updates must be *efficient*. The interruption of (part of) the system service (usually called *disruption*) must be minimal as well as the delay with which the system is updated (called *timeliness*).

This paper addresses the timeliness and disruption of dynamic changes in Component-Based Distributed Systems (CBDSs), which have already been widely studied in the past [1], [2], [3]. Kramer and Magee [1] proposed a seminal solution where components can only be updated when they reach a *quiescent* status. To ensure the consistency of an update, this approach preventively blocks all computations that may traverse the components affected by the reconfiguration. This is achieved by analyzing the *static* dependencies between the components of the distributed architecture. The approach is conservative and may bring more disruption than necessary. Since static dependencies pessimistically include all the *potential* interactions between components, disruption might be mitigated by considering *dynamic* dependencies. These are temporal relationships between components caused by on-going computations, and they only indicate the *current* constraints on the reconfigurability of the system. At runtime, this information helps identify a less conservative condition that reduces the degree of disruption and improves the timeliness of the update.

In [4] we proposed *version consistency* as a criterion for the safe and efficient dynamic updates of the components of transactional CBDSs. We introduced a condition that only checks the dynamic dependencies on the components-to-update to ensure their correctness. Other approaches, like the one based on *tranquillity* [3], and those presented in [5] and [6], use dynamic dependencies to only ensure the consistency of transactions and of their direct sub-transactions, which may lead to unsafe updates. In contrast, our criterion exploits dynamic dependencies to ensure the consistency of transactions, along with their direct and indirect sub-transactions.

Version consistency can be used to update existing components at runtime. As the other approaches, it assumes that the old and new versions of components be correct and that updates be atomic and instantaneous (no delays and interleaving are taken into account) to avoid considering the possible internal states of components during the

- *L. Baresi and C. Ghezzi are with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano 20133, Italy. E-mail: {luciano.baresi, carlo.ghezzi}@polimi.it.*
- *X. Ma is with the State Key Laboratory for Novel Software Technology and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University, Nanjing, Jiangsu 210093, China. E-mail: xxm@nju.edu.cn.*
- *V. Panzica La Manna is with Holst Centre/imec the Netherlands, Eindhoven 5656 AE, The Netherlands. E-mail: valerio.panzicalamanna@imec-nl.nl.*

Manuscript received 8 Oct. 2014; revised 2 July 2016; accepted 4 July 2016. Date of publication 18 July 2016; date of current version 24 Apr. 2017. Recommended for acceptance by M. Dwyer.

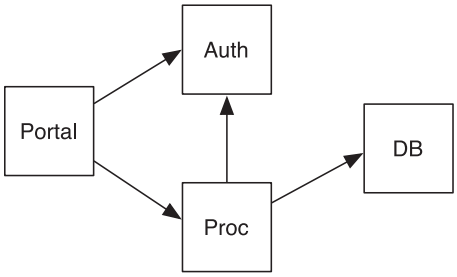


Fig. 1. Components of the running example.

update. The safety of all these approaches relies on the basic design principles of isolation and consistency [7], that is, the executions of the different distributed transactions do not interfere with one another.

As the other approaches, version consistency also assumes that the system architecture does not change, that is, static dependencies are given, while new versions of components can change their behavior, and thus dynamic dependencies can vary while the system executes. The paper only considers that single components be changed, but the extension to multiple components is trivial: atomic updates can deal with sets of components-to-update seen as single "virtual" components.

Since version consistency exploits dynamic dependencies, it requires information about the execution logic of components to achieve better efficiency. Our prototypical framework shows how this information can be retrieved automatically, without posing additional burden on the software engineer.

In summary, version consistency ensures the same guarantees as the quiescence-based solution [1], but with less disruption and more timeliness. In contrast, it extends the guarantees provided by the tranquillity-based solution [3] to the case of multi-party distributed transactions. If the system does not comprise this kind of transactions, the tranquillity-based solution provides enough guarantees and, being narrower in scope, is lighter than version consistency.

This paper[1] extends our previous work in the following directions:

- We polish the proposed approach and formalize it by means of a graph transformation (GT) system [8];
- We use GROOVE [9] as graph transformation modeling and verification tool to prove the correctness of our approach and to implement Version Consistency Checker (VCC), a viable solution for reasoning on dynamic updates;
- We present ConUp, a novel implementation of the approach based on Service Component Architecture (SCA), [10];
- We evaluate the approach and ConUp on a significant, third-party example application to show the benefits in terms of disruption and timeliness with respect to the quiescence-based solution.

The rest of the paper is organized as follows. Section 2 introduces a running example used to illustrate our proposal and to highlight the limitations of the two reference approaches in the field. Section 3 describes our model of CBDS and the challenges posed by dynamic updates. Section 4 presents version consistency and the management framework. Section 5 proposes the algorithm for the distributed management of dynamic dependencies. Section 6 describes how we used a graph transformation system to model the algorithm and verify its correctness through model checking. Section 7 introduces ConUp, our prototypical framework for managing the dynamic updates (version consistency) of SCA-based systems. Section 8 summarizes the assessment we conducted. Section 9 surveys related approaches and Section 10 concludes the paper.

1. The original technical report, the graph transformation rules we defined, ConUp, VCC, and more data about the evaluations we carried out are available here: https://github.com/brickinwall/conup

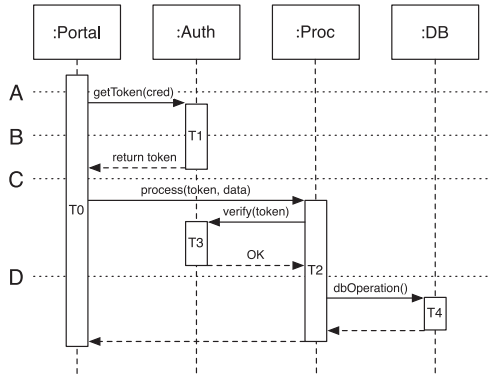


Fig. 2. Detailed scenario.

2 Running Example

Let us consider a simple web system whose components and interconnections are shown as graph nodes and edges in Fig. 1. A portal component (Portal) interacts with an authentication component (Auth) and a business processing component (Proc), while Proc interacts with both Auth and a database component (DB).

Fig. 2 shows a detailed usage scenario to exemplify the dynamic dependencies required by our approach. The complete, explicit elicitation of the distributed transactions that may be run on the system is not required by our approach. The sequence diagram is provided here only to focus on a concrete intuitive case. Transactions are described by rectangles $T_0 \cdots T_4$: Portal first gets an authentication token from Auth and then uses it to require the service from Proc. Proc verifies the token through Auth and then starts computing and interacting with DB.

Let us now suppose that Auth be updated to exploit a stronger encryption algorithm. Although the new algorithm is incompatible with the old one, the other components remain unaware of it because all encryption/decryption operations are done within Auth. The problem is that if we update Auth at runtime, we must ensure that all running transactions execute correctly before, during, and after the update. If the update happens at any time, it is impossible to ensure correctness automatically [11].

An obvious restriction on *when* the update can happen is to impose that the component targeted for update be *idle*, that is, the component is not busy executing transactions. However, being idle is often insufficient for a component to be safely updated. For example, considering the scenario of Fig. 2, an update of Auth at time **C**, i.e., when the component

is idle, would be unsafe since the security token would be created with an algorithm and validated by another.

The *quiescence*-based approach proposed by Kramer and Magee [1] states a sufficient condition for a node to be safely manipulated in dynamic reconfigurations. A node cannot be quiescent before the completion of all the transactions initiated by dependent nodes. This means that the actual update could be deferred significantly. In our example, Auth cannot be quiescent before the end of the transactions hosted on Portal and Proc (T_0 and T_2). Moreover, the progress of all the other nodes that could potentially initiate transactions, which require service from Auth, directly or indirectly, has to be blocked till the end of the update. Again, in our example Portal and Proc cannot proceed before changing Auth. This means that the adoption of this approach could introduce significant disruption in the service provided by the system.

To reduce disruption, Vandewoude et al. [3] proposed the concept of *tranquillity*, as alternative to quiescence. The idea is that there is no need to wait that a transaction completes its execution if it will not require the service provided by the component-to-update anymore, even if the component was involved in the transaction. It is also permitted to update a node even if some on-going transactions will require the service provided by the node in the future, but they have not interacted with it yet. While claimed to be "a sufficient condition for application consistency during a dynamic reconfiguration" [3], the notion of tranquillity is based on a rather strong assumption. In fact, a distributed transaction can only contain a root transaction and the sub-transactions directly initiated by the root transaction. A sub-sub-transaction (i.e., a sub-transaction initiated by another sub-transaction) must be independent of the root transaction, and can thus use any version of a component. In [3] the authors argue that this assumption is correct for systems made of reusable components that follow the black-box principle. However, as witnessed by our running example, and later also by the example application used in Section 8, this assumption does not hold in many practical scenarios. In addition, it can be difficult for a software engineer to decide whether this assumption holds or not. Incautious applications of the approach to the scenario of Fig. 2 would permit unsafe updates. In fact, after Auth returns the token to Portal, it does not participate in the session initiated by Portal anymore. In addition, before the request for verification is sent, Auth has not participated in the session initiated by Proc yet, and thus Auth is tranquil at time **C**. However, if Auth were updated at this time, the verification would fail because the token was issued by the old version of Auth—with a different encryption algorithm. This failure would not happen if the system entirely complied with either the old or the new system configuration.

To conclude, we can say that the quiescence-based approach provides a general and safe solution, but it can be highly disruptive. The tranquillity-based approach is less disruptive, but its assumption is too restrictive to be applicable to a wide set of systems (e.g., our example). Our goal is to get the best of the two proposals and add efficiency and timeliness to safety.

3 Problem Setting

This section presents the definitions required to introduce version consistency. We first describe both the architectural and computational models of a component-based distributed system. We then define when a dynamic update is correct and prove that, for an arbitrary update, we cannot derive a necessary and sufficient condition that ensures its correctness.

3.1 Preliminary Definitions

The *static configuration* of a component-based distributed system is defined as follows:

Definition 1 (Static Configuration). *A static configuration of a component-based distributed system is a directed graph whose nodes represent versioned components. A directed edge from a given node c_1 to another node c_2 is called static edge and represents a static dependency, that is, the possibility for c_1 to require a service provided by c_2.*

Besides the static configuration, our approach requires information about the dynamic interactions between components. To this end, a *transaction* is a sequence of actions that accomplishes a given task and completes in bounded time. Actions include local computations and message exchanges.

Our notion of transaction is weaker than Atomic, Consistent, Isolated, and Durable (ACID) transactions of database systems [7]. Since this paper focuses on the consistency of dynamic updates, we only assume that all local transactions are consistent and isolated. This assumption generally applies to existing component-based frameworks such as the ones based on Enterprise Java Beans [12] and Web Services [13].

A transaction T, executed on component h_T, can be initiated by an external client or by another transaction T'. T is called a *root transaction* in the former case and a *sub-transaction* (of T') in the latter case. A root transaction has a unique, system-wide identifier. For each transaction T, h_T must know $root(T)$, which is the identifier of the root transaction of T. For example, T_4 is a sub-transaction of T_2 in the sequence diagram of Fig. 2. The term $sub(T, T')$ denotes that T' is a direct sub-transaction of T. A transaction can be a direct sub-transaction of at most one transaction,[2] and by definition a root transaction is not a sub-transaction of any other transaction. All the messages exchanged between a transaction T and its sub-transaction T' are temporally scoped between the two corresponding messages that initiate T' and notify its completion. We also assume that transactions are always notified of the completion of their sub-transactions and that a transaction cannot end before the termination of its sub-transactions.

The set $ext(T) = \{x | x = T \vee sub^+(T, x)\}$ is the *extended transaction set* of T, which contains T and all its direct and indirect sub-transactions. The extended transaction set of a root transaction models the concept of *distributed transaction* that can span over multiple components. For example, if we consider the root transaction T_0 on Portal, its extended transaction set is $ext(T_0) = \{T_0, T_1, T_2, T_3, T_4\}$, where T_1 on

2. This constraint refers to the relationships between running entities, not to their definitions.

Auth is in response to the getToken request, T_2 on Proc in response to process, T_3 on Auth in response to verify, and T_4 on DB for T_2's request of database operations.

3.2 Correct Dynamic Updates

The dynamic update of a single component can be defined as follows:

Definition 2 (Dynamic Update). *A dynamic update of a component c can only occur when c is idle. The update is specified as a tuple $\langle \Sigma, c, c', \mathcal{T}, s \rangle$ where Σ is the original system configuration, c is a component of Σ to be replaced with a new version c', s is the current state of component c and $\mathcal{T}$ is a state transfer function. $\mathcal{T}(s)$ yields the initial state of c' after replacement.*

Note that the definition takes into account the general case in which components have an internal state.[3] Should components be purely functional, no state transformation function would be needed.

As already stated at the beginning of the paper, since we assume that for a given dynamic update $\langle \Sigma, c, c', \mathcal{T}, s \rangle$, the corresponding off-line[4] update $\langle \Sigma, c, c' \rangle$ be correct, the transactions running on Σ satisfy the old system specification $\mathbb{S}$ and the transactions on $\Sigma' = \Sigma[c'/c]$ satisfy the new specification $\mathbb{S}'$. Given the distributed nature of these transactions we can only adopt a *weak* definition of *correctness* of dynamic updates, which is defined as follows:

Definition 3 (Correct Dynamic Update). *A dynamic update $\langle \Sigma, c, c', \mathcal{T}, s \rangle$ is correct iff:*

- *The transactions that end before the update satisfy $\mathbb{S}$;*
- *The transactions that begin after the update satisfy $\mathbb{S}'$;*
- *The transactions that begin before the update, and end after it, satisfy either $\mathbb{S}$ or $\mathbb{S}'$.*

Note that the first bullet is implied by the correctness of the corresponding off-line update, while the other two bullets add the runtime dimension. This definition of correctness also applies to the quiescence-based approach; the idea is to propose a more efficient way to enforce it, not a new definition of correctness.

In general, we cannot have an automatically checkable condition that is sufficient and necessary for the correctness of a dynamic update that occurs at any time [11]. This is true even if the corresponding off-line update is guaranteed to be correct and the dynamic update only happens when the component is idle.

Proposition 1. *Given an arbitrary dynamic update $\langle \Sigma, c, c', \mathcal{T}, s \rangle$, such that the corresponding off-line update $\langle \Sigma, c, c' \rangle$ is correct and component c is idle in state s, the correctness of the dynamic update is undecidable.*

Proof. Let us assume a system that consists of two components A and B. A provides a function f whose implementation uses a function g provided by B. The specification of the system is that $A.f()$ returns 0.

```
Component A:
int f() { return B.g(B.g(2)); }
Component B:
int g(int x) { return 0; }
```

Now component B is about to be dynamically updated with B' that provides g'. The new system specification is the same as the old one. Let us suppose there is an algorithm $\mathcal{D}$ that can decide, for an arbitrary g' that ensures that the off-line update be correct, whether the dynamic update that happens when B is idle is correct or not. Then we could use $\mathcal{D}$ to decide whether an arbitrary program $h()$ will eventually halt as follows. Let us construct g' as

```
int g'(int x) {
  switch (x){
   case 2: return 1;
   case 1: return 0;
   default: { h(), return 0 };
  }
}
```

Note that g' ensures the correctness of the off-line update because $B'.g'(B'.g'(2))$ returns 0. Let us consider that the update happens after $A.f$ calls $B.g$ the first time but before the second time. In that moment, B is idle if the transaction were the only one in the system. In this situation $\mathcal{D}$ must be able to decide whether $B'.g'(B.g(2))$ returns 0. Since $B.g(2)$ returns 0, the result of $\mathcal{D}(g')$ is the same as whether $h()$ will eventually halt. Because whether an arbitrary program will eventually halt is undecidable, such an algorithm $\mathcal{D}$ cannot exist. The correctness of dynamic updates is then generally undecidable because at least it is undecidable in this special case.

4 Version Consistency

Since the correctness of arbitrary dynamic updates is undecidable, we can only derive some automatically checkable conditions that are sufficient for the correctness by constraining *when* the runtime update can be performed. These conditions must be: (a) strong enough to ensure the correctness of dynamic updates, (b) weak enough to allow for efficient and timely changes, and (c) automatically checkable in a distributed setting (without enforcing unnatural centralized solutions).

To this end, the paper proposes the notion of version consistency as a sufficient condition for the correct and efficient dynamic update of CBDSs:

Definition 4 (Version Consistency). *Transaction T is version consistent with respect to an update $\langle \Sigma, c, c', \mathcal{T}, s \rangle$ iff $\nexists T_1, T_2 \in ext(T) \mid h_{T_1} = c \wedge h_{T_2} = c'$. A dynamic update $\langle \Sigma, c, c', \mathcal{T}, s \rangle$ is version consistent if c is idle in s and all the transactions in the system are version consistent.*

Given the correctness of the off-line update and the independence among distributed transactions, this condition is sufficient for the correctness of dynamic updates. This is because any extant transaction with all its (direct and indirect) sub-transactions is entirely executed in the old or in the new configuration. Also note that a transaction that

3. The state of a component is nothing but a set of values stored within the component that: (a) are properly initialized when the component is enacted, (b) are accessed and modified by operations executed within the component, (c) are always accessed atomically, and (d) can persist the execution of different (distributed) transactions.

4. The difference between a dynamic update and its corresponding off-line update is that the former is carried out while the system is in operation; the latter after its shutdown.

ends before (starts after) the update cannot have a sub-transaction hosted on the new (old) version of a component being updated.

Back to our example, if the update of Auth happens after transaction T_0 begins but before it sends a getToken request to Auth (time **A**), all the transactions in $ext(T_0)$ (i.e., all the transactions in Fig. 2) are served in the same way as if the update happened before they all began. If it happens at any time after Auth replies to the verify request issued by Proc (time **D**), all the transactions in $ext(T_0)$ are served in the same way as if the update happened after they all ended. However, if it happens at time **C**, then $h_{T_1} = \mathsf{Auth}$, but $h_{T_3} = \mathsf{Auth}'$. As both T_1 and $T_3 \in ext(T_0)$, T_0 would not be version-consistent.

4.1 Dynamic Dependencies

Since the definition of version consistency predicates on distributed transactions, it cannot be checked on single components. We need to identify a condition that is checkable locally, that is, on the component(s)-to-update, and yet ensures version consistency.

To define such a condition, we introduce the notion of dynamic dependency, represented by properly labelled edges added to the static configuration of the system. Dynamic edges are added and removed on demand and they are labelled with either f (future) or p (past). A future edge represents the possibility for the source node to initiate a transaction T on the target node; a past edge witnesses the fact that a transaction T initiated by the source node has already been executed on the target node. Future and past edges are also labelled with the identifier of the root transaction of T. We use $c_1 \xrightarrow{f/T} c_2$ ($c_1 \xrightarrow{p/T} c_2$) to denote a future (past) edge, labelled with the identifier of root transaction T, from component c_1 to component c_2.

Definition 5 (Valid Configuration). *A static configuration decorated with* past/future *edges (hereafter, a configuration) is valid if these edges are created and removed at runtime according to the following constraints:*

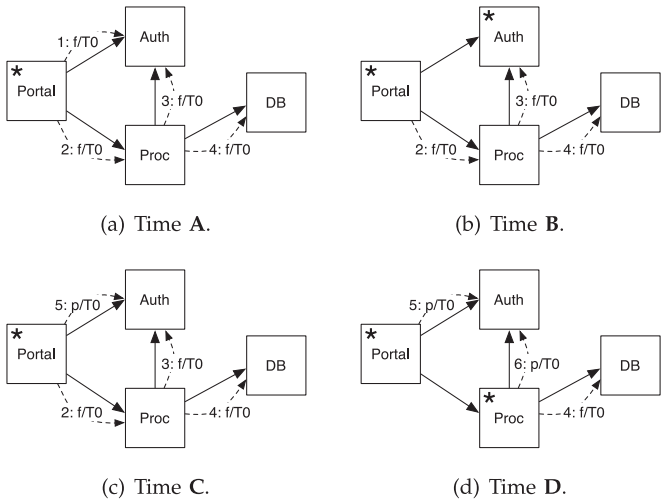
(1) (LOCALITY) *Any* future *or* past *edge between two nodes cannot be created if there does not exist a* static *edge that connects the same two nodes;*

(2) (FUTURE-VALIDITY) *A* future *edge* $c_1 \xrightarrow{f/T} c_2$ *must be added before the first sub-transaction* $T' \in ext(T)$*, where* $T' \neq T$*, is initiated and it cannot be removed as long as transactions hosted on* c_1 *may initiate further* $T'' \in ext(T)$ *on* c_2*;*

(3) (PAST-VALIDITY) *A* past *edge* $c_1 \xrightarrow{p/T} c_2$ *must be added at the end of the first transaction* $T' \in ext(T)$ *on* c_2*, initiated by a transaction hosted on* c_1 *and cannot be removed at least until the end of* T*.*

Fig. 3 shows some configurations of the example system. Active components, that is, components that are executing transactions in $ext(T_0)$, are marked with a $*$, and numbers mimic the order with which edges are added.

The configuration of Fig. 3a corresponds to time **A** in Fig. 2: transaction T_0 is executing on Portal, which is $*$-annotated. The dynamic edges indicate that to serve transactions in $ext(T_0)$, Portal might use Auth and Proc in the future, and also



(a) Time **A**. (b) Time **B**. (c) Time **C**. (d) Time **D**.

Fig. 3. Some configurations of the example system with explicit dynamic dependencies.

Proc might use Auth and DB. Fig. 3b corresponds to time **B** and says that a transaction in $ext(T_0)$ (T_1) is currently running on Auth, but no further transaction in $ext(T_0)$ hosted on Portal will initiate any sub-transaction on Auth anymore because there is no T_0-labelled future edge between the two nodes. Fig. 3c, which corresponds to time **C** in Fig. 2, indicates that Auth might have hosted transactions in $ext(T_0)$ initiated by Portal in the past, and might host further transactions in $ext(T_0)$ initiated by Proc in the future. Fig. 3d corresponds to time **D** in Fig. 2 and shows that Auth, although it might have hosted transactions in $ext(T_0)$, is not hosting and will not host these transactions anymore.

Note that to keep a configuration valid, and consider alternative execution flows, one can adopt a conservative management of future/past edges between components. For example, if T_2 on Proc needs to evaluate the results from DB before deciding whether to ask Auth for verification, one can keep the future edge from Proc to Auth till T_2 does not need Auth anymore. In the worst case, it could be kept till the end of T_2. This flexibility makes our approach generally applicable in different situations with diverse degrees of accuracy.

4.2 Freeness

Given a valid configuration, we can identify a locally checkable sufficient condition for the version consistency of dynamic updates, called *freeness*:

Definition 6 (Freeness). *Given a configuration* Σ*, a component* c *is said to be* free *of dependencies with respect to a root transaction* T *iff* c *is not hosting any transaction in* $ext(T)$ *and there does not exist a pair of* T*-labelled future/past edges entering* c*.* c *is said to be free in* Σ *iff it is free with respect to all the root transactions in the configuration.*

In our example, Auth is free with respect to T_0 in the configurations of Figs. 3a and 3d, but not in the one of Fig. 3c since there exist two edges f/T_0 and p/T_0 that enter Auth. Moreover, since Auth is active in Fig. 3b, this trivially falsifies its freeness. Intuitively, for a valid configuration Σ, the freeness condition for a component c—with respect to a root transaction T—means that the distributed transaction modeled by $ext(T)$ either has not used c yet (otherwise there

should be a past edge), or it will not use c anymore (otherwise there should be a future edge). This leads to the following proposition.

Proposition 2. *Given a valid configuration Σ of a system, a dynamic update of a component c is version consistent if it happens when c is free in Σ.*

Proof. By contradiction, let us assume that Σ is valid, c is free, but version consistency does not hold. Because c is free, it must be idle when the update occurs. So there must be a transaction T that is not version consistent, that is, $\exists\ T_1, T_2 \in ext(T) \mid h_{T_1} = c\ \wedge\ h_{T_2} = c'$.

First, T is not hosted on c. Because T begins no later than T_1 and T_1 begins before the update since $h_{T_1} = c$, T begins before the update. Similarly T ends after the update because of T_2. However c is idle when the update happens, hence $h_T \neq c$.

Second, there is a T-labelled past edge entering c. Considering the sub-transaction chain from T to T_1, there must exist T_i, T_j such that $sub(T_i, T_j) \wedge h_{T_i} \neq c \wedge h_{T_j} = c$. According to the past-validity of Σ and the fact that T_j must have already ended when the update happens, such a past edge must exist.

Third, there is a T-labelled future edge entering c. Without any loss of generality, let T_2 be the first transaction in $ext(T)$ initiated on c'. Let us consider the sub-transaction chain from T to T_2. Although some of the ancestor transactions of T_2 may run *after* the update happens, their behavior is *not* changed by the update because they are independent of transactions not in $ext(T)$. Thus at the time of the update, c is expected to host transactions in $ext(T)$ in the future. According to the future-validity of Σ, there must be a T-labelled future edge entering c that has already been created and has not been removed yet.

So there is a pair of T-labelled past/future edges entering c. This contradicts the freeness of c.

5 Dynamic Dependencies

The definition of *valid configuration* constrains the time intervals during which dynamic edges should exist to keep a configuration valid. One can satisfy all the conditions straightforwardly by creating all the future and past edges at the beginning of a root transaction and by removing them at the end of it. Although version consistency would be ensured, disruption could be excessive.

Our solution proposes a distributed algorithm for efficiently managing dynamic dependencies that: (1) keeps the configuration valid and (2) ensures version consistency with limited disruption. We assume that components be split into two parts: *application logic* and *container*. The former must be provided by the implementor, while the latter is added automatically by the underlying middleware infrastructure and is responsible for managing dependencies. Although our solution leverages some information from application logic, containers are transparent to it and keep the separation between application and adaptation logic—as proposed by the other approaches.

Dynamic dependencies are maintained in a distributed way. Each component only has a local view of the configuration that includes itself and its direct neighbors. A component is responsible for the creation and removal of the outgoing dynamic edges, but it is also always notified of the creation and removal of the incoming ones. This is achieved by exchanging management messages that keep the consistency among the views of neighbor components.

The management of dynamic dependencies may slightly delay the execution of the actual transactions, but it guarantees that no transaction will be blocked forever. The underlying message delivery is assumed to be reliable, and the messages between two components are kept in order. Dynamic edges are labelled with the identifiers of the corresponding root transactions to allow for the management of the dynamic edges of a root transaction independently of those of other transactions.

We require that given a transaction T, its host component h_T always knows $f(T)$, the set of static edges through which it might initiate sub-transactions on neighbor components in the future, and $p(T)$, the set of static edges through which it has initiated sub-transactions in the past. It is safe to over estimate $f(T)$ and $p(T)$, but better accuracy means better timeliness and less disruption in dynamic updates. This information can be derived from design documents, be obtained by monitoring the execution, or even be automatically extracted from the implementation of application components, as exemplified by our ConUp framework (Section 7). The same is required for deciding the *tranquillity* of components [3].

If we consider a distributed transaction $ext(T)$, our algorithm consists of three steps. **Setup** is carried out after the root transaction T is initiated and before it initiates any sub-transaction. During this phase, h_T creates a future edge for each of its out-going static edges that T might use to initiate sub-transactions according to $f(T)$, notifies the corresponding neighbor components, and waits for their acknowledgements. Only after receiving all these acknowledgements, T is allowed to initiate its sub-transactions.

Whenever a component c_1 is notified of the creation of an incoming future edge by c_2, it creates new future edges towards its statically-dependent components. Again, it notifies the neighbor components, waits for their acknowledgements, and then acknowledges c_2 back. The rationale is that by accepting the creation of a future edge, c_1 "promises" c_2 to host some $T_{c_1} \in ext(T)$ in the future, but to make such a promise c_1 first needs to get the promises from those components that T_{c_1} might need to use. Loops in the static configuration are handled by avoiding creation and notification of duplicated future edges. This conservative way creates future edges to achieve a valid configuration with respect to T when the set up step finishes. For example, Fig. 3a shows the result of setting up future edges for transaction T_0 of Fig. 2.

While in **Progress**, the transactions in $ext(T)$ execute, past edges are created to register *have-used* relationships, and future edges are gradually removed as soon as a component *will-not-use* another one anymore. To record the *have-used* information, when a sub-transaction T' originally initiated by T ends, the corresponding past edge is created *immediately*. This is done by letting $h_{T'}$ first notify h_T the end of T', and by keeping T' alive till $h_{T'}$ is notified of the creation of the edge.

In our example scenario, Portal removes the future edge to Auth after it initiates T_1 on Auth as T_0 will not initiate

such transactions anymore (Fig. 3b). When T_1 ends, Portal immediately creates a past edge to record the fact that it has used Auth (Fig. 3c). Eventually, the system reaches the configuration of Fig. 3d, where Auth is free with respect to T_0.

Cleanup is carried out only when T ends. The algorithm recursively removes all remaining past and future edges. This step does not affect the validity of the configuration.

5.1 Achieving Freeness

Checking a valid configuration for freeness is straightforward since the condition is local to the component that is to be updated, but freeness can be achieved in different ways.

The first strategy, called *waiting for freeness (WF)*, is simply opportunistic. The system just waits for freeness to manifest itself. This strategy has no extra overhead other than setting up and maintaining the valid configuration and checking for the freeness of the component. However, although every transaction completes in finite time, the freeness of a component could never be reached—for example, there could always be transactions running on it.

To improve timeliness, other strategies may be devised. The strategy called *concurrent versions (CV)* lets two versions c and c' of a component co-exist during the update. The use of multiple versions of the same component is not new (e.g., HERCULES [14] and Upstart [15] adopt it), but our proposal adds version consistency as a means to choose which version must serve a request and to decide when a version can be decommissioned. Given a valid configuration, one can choose component c to serve the requests that come from a transaction T only if c has already an incoming past edge labeled with $root(T)$, or c is currently hosting a transaction T' with $root(T') = root(T)$, and let c' serve the others. This means that c cannot have incoming past edges labelled with new root transactions, and it will eventually become free since old transactions will reach a stage where they will not use it anymore. As soon as this component becomes free, it can be safely removed from the system.

However, if multiple versions cannot coexist, e.g., due to limited resources, another strategy called *blocking for freeness (BF)* can be adopted. It requires that some of the requests to component c be temporally blocked to avoid creating past edges labelled with new root transactions. This component will eventually become free when all existing transactions do not use it anymore. More precisely, the initiation of any transaction on component c is blocked unless it belongs to an extended transaction set in which a member transaction has already been hosted on c. This comes from the fact that the first T-labelled past edge entering c is created the first time a transaction in $ext(T)$ is being initiated on c. As soon as c becomes free, it is substituted by the new version c', and all blocked transactions are resumed.

Back to our example, if Auth receives the request for update before T_0 on Portal initiates T_1 on Auth, the initiation of T_1 is blocked because there is no T_0-labelled past edge entering Auth. Note that there can be T_0-labelled future edges entering Auth (e.g., see Fig. 3a), but by blocking the initiation of any $T \in ext(T_0)$ on Auth the freeness of this component will not be hindered by T_0 since no T_0-labelled past edge needs to be created. However, if the request for update arrived at time **C**, the initiation of T_3 by T_2 on Proc would be allowed because there is already a T_0-labelled past edge entering Auth (created when T_1 ended). In this case, Auth must wait for all T_0-labelled future edges to be removed, that is, at time **D**, which corresponds to the configuration presented in Fig. 3d.

This last strategy relies on the assumption that the progress of a transaction in an extended transaction set will not be blocked by a transaction in another extended transaction set—otherwise, we may have deadlocks. For example, this happens when two transactions are concurrently hosted on the same component and need to access some shared resources. If one blocked the transaction that holds the lock (to avoid creating new past edges to the component-to-update), the other would be blocked as well (since it does not hold the lock), blocks would never be released, and the system would enter a deadlock.

In the case of dependent distributed transactions, one can rely on existing mechanisms for the avoidance, prevention, or detection of distributed deadlocks. For example, according to the Wait-for Graph used for deadlock detection [16], one can resume the initialization of blocked transactions if their initializers are waited for by some transactions that must proceed to free shared resources. Note that such a mechanism is very likely already provided by the system infrastructure since distributed deadlocks are an important, potential problem in distributed systems. One can also use a more conservative blocking policy, and delay the creation of future (past) edges labelled with new root transactions. This method prevents the initialization of any new root transaction that may use the targeted component, thus introducing more disruption.

Generally, strategy CV is preferred when applicable. Strategies WF and CV do not introduce disruption other than that caused by setting up and maintaining the valid configuration, while strategy BF imposes extra disruption due to its temporal blocking of some transactions. As for timeliness, strategies CV and BF are essentially equivalent. BF should then be used when CV is not applicable, for example because of constraints on deployed components. CV is preferable every time there are no constraints on the number of active components, on their deployment, and on the policies for switching from one version to another.

5.2 On-Demand Setup

The above algorithm for managing dynamic dependencies assumes that configurations be always kept up-to-date no matter whether there is any request for update. However, configuration management does not come for free. If updates are rare, a valid configuration can be set up on demand only when a request is planned or expected.

Establishing the validity of the configurations of a root transaction is not difficult, as we can safely create enough future/past edges. However, the creation of these edges must be coordinated with the execution of existing transactions, and the initialization of new ones. These activities must take into account all the transactions that could be affected by the update, but they must also avoid unnecessary constraints (serialization) among unrelated ones.

To this end, each component is associated with one of the following *working modes*: NORMAL, ONDEMAND, or VALID. NORMAL means that the component is not managing dynamic dependencies, VALID means that it is managing

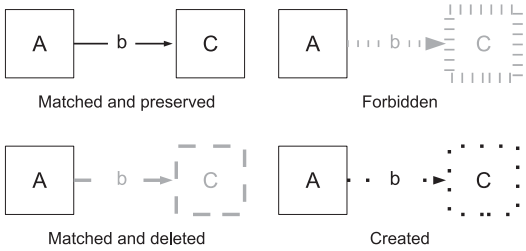


Fig. 4. Elements of GROOVE graph transformation rules.

dynamic dependencies, and the validity of the configuration has been established for all the distributed transactions it is involved in, and ONDEMAND is an intermediary mode. It imposes that the component: (1) manages the dynamic dependencies for all the new root transactions that are initiated locally, (2) blocks the initiation and termination of locally hosted sub-transactions temporarily, and (3) for each locally-hosted ongoing root transaction T, creates future (past) edges towards the components that might host in the future (might have hosted in the past) transactions in $ext(T)$ in a way similar to the one described above.

At the beginning, all components operate in mode NORMAL. When a component receives a request for update, it starts the process and waits for its mode to become VALID before using the proposed approach for achieving freeness. Upon receiving a request for set up, a component c switches to mode ONDEMAND and sends set up requests to all the components c_i that statically depend on it. Once c has finished the set up of dynamic edges for its local root transactions, and has received all the acknowledgements from c_i, it switches to mode VALID, resumes all blocked transactions, and the validity of the configuration is established. Then, c can be updated once its freeness is achieved.

6 Specification and Verification

Our formalization exploits a graph transformation system to model the proposed algorithm for managing dynamic dependencies and to verify its correctness. The graph transformation system is also the basis of Version Consistency Checking, a tool for reasoning on the dynamic update of CBDSs.

We decided to use a graph transformation system because we wanted to specify the different steps of our solution and graph transformations provide a formal, concise, and executable means to specify the creation and evolution of graph-like structures. They have been used in the past to model, among others, software architecture [17], distributed systems [18], and visual languages [19]. The analysis capabilities provided by tools like GROOVE [9] and AGG [20] also allow for the formal verification of significant properties of designed systems. It is true, however, that all these verification tools must cope with the intrinsic exponential growth of the problems at hand [21], and in many cases they can only address partial, or simplified ones. We do not have investigated different verification solutions since our goal was mainly the formalization of the approach, and the (partial) verification of designed systems was almost a side effect.

A graph transformation system can be used to describe the evolution of distributed entities: nodes represent the different elements and graph transformation rules define how pieces (entities) of the whole system can evolve. Different

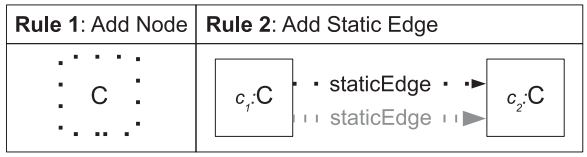


Fig. 5. GT rules for creating static configurations.

rules can be applicable at the same time non-deterministically and in some cases the application of a rule may preclude the execution of another. There is thus a dichotomy between the centralized nature of a graph and the local nature of the different rules that govern its evolution, thus mimicking the behavior of a distributed system where different parts can evolve independently and in parallel.

In this paper we use GROOVE as modeling and verification tool. A GROOVE graph transformation rule can include the following elements (visually summarized in Fig. 4): (i) a sub-graph (i.e., nodes and edges) that must exist in the target (host) graph, and that remains untouched, to enable the application of the rule (continuous black line); (ii) a sub-graph that must not exist in the host graph to enable the application of the rule (grey vertical lines); (iii) a sub-graph that will be deleted from the host graph by applying the rule (grey dashed line); and (iv) a sub-graph that will be added to the host graph by applying the rule (black dotted line).

The rules presented in the rest of the paper are based on the following node and edge types. As for nodes, types **C** and **T** correspond to components and transactions, while **F** and **P** mimic future and past edges. **FCreated** and **PCreated** are used to render the cases in which an **F** or **P** node has already been created, and **THostedOnNewVersion** and **THostedOnOldVersion** to identify the version of a component on which a transaction is running. The meaning of edges is the following: **staticEdge**, **in**, **out**, **old**, and **new** are self-explanatory. **hosted** and **subTx** help understand the component on which a transaction is running and its possible sub-transactions, while **mayUseInFut** is used to identify the transactions that may use a given component in the future. A *mayUseInFut* edge from a transaction T to a component c models the potential future initiation of sub-transactions on c by T; its absence indicates that T will not use c anymore. In other words, these edges materialize sets $f(T)$: for each static edge from h_T to a neighbor component c in $f(T)$, there exists a *mayUseInFut* edge in the graph from T to c. Since the graph transformation system has no knowledge of the internals of components, it can only abstract them as creations and removals of *mayUseInFut* edges, whose corresponding rules are trivial and omitted here.

During verification, we create a *mayUseInFut* edge for each outgoing static edge of the component of interest. This way $f(T)$ is initialized with all possible static edges. A second rule removes *mayUseInFut* edges, one by one, and it is applied non-deterministically. This non-determinism is used to mimic any possible dynamic evolution of a transaction that starts using a component and then, given its internal evolution, does not use it anymore.

Rule 1 and **Rule 2** of Fig. 5 allow for the generation of any arbitrary static configuration. **Rule 1** states that it is always possible to add a new component. **Rule 2** describes the creation of static edges. A static edge can be created if there exist two distinct components (positive application

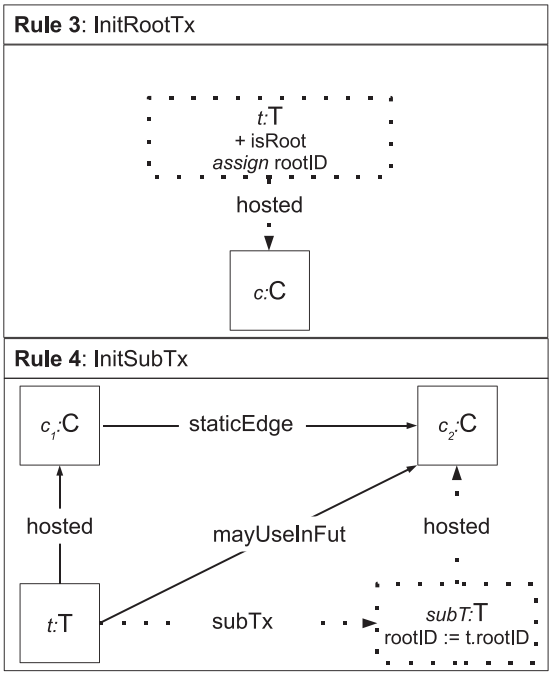


Fig. 6. GT rules for creating transactions.

condition) and they are not already connected. Needless to say, the configuration of Fig. 1 can be created by applying **Rule 1** and **Rule 2** four times each.

The notions of transaction, root transaction, and sub-transaction are formalized by **Rule 3** and **Rule 4** of Fig. 6. **Rule 3** creates a root transaction on a given component and assigns it a unique identifier *rootID*. **Rule 4** formalizes the fact that a sub-transaction can be initiated by a transaction t hosted by component c_1 onto a component c_2 if there exists a static edge between c_1 and c_2 and a *mayUseInFut* edge between t and c_2. If the conditions specified by **Rule 4** are met, a new transaction (of type *T*) is created with the same *rootID* as the initiating transaction, and a new edge, named *subTx*, states that the initiated transaction is a sub-transaction of *T*.

Rule 5 of Fig. 7 describes the creation[5] of a future edge from the component that initiated the root transaction c_1 to a statically dependent component c_2. Besides creating an f edge, it also adds an *FCreated* node to mark the existence of the edge. This node is added when the f edge is created, and removed at the end of the distributed transaction. Note that *FCreated* (and *PCreated*) nodes are placeholders needed for verifying the algorithm. They are added the first time the f/p edge between two components is created, and used to know whether the edge has already been created, and maybe removed, or it has never existed.

Whenever a component c_1 is notified of the creation of an incoming future edge, it creates new future edges towards its statically-dependent components (obtained by repeatedly applying **Rule 6**).

Rule 7 of Fig. 8 models the creation of a past edge—along with a *PCreated* node—just after the termination of the first sub-transaction *subT* initiated by c_1 on c_2. **Rule 8** models the termination of the other sub-transactions when a past edge with the same *rootID* between the two nodes already exists. This rule only removes node *subT* and the associated

5. Acknowledgements are left implicit for the sake of simplicity.

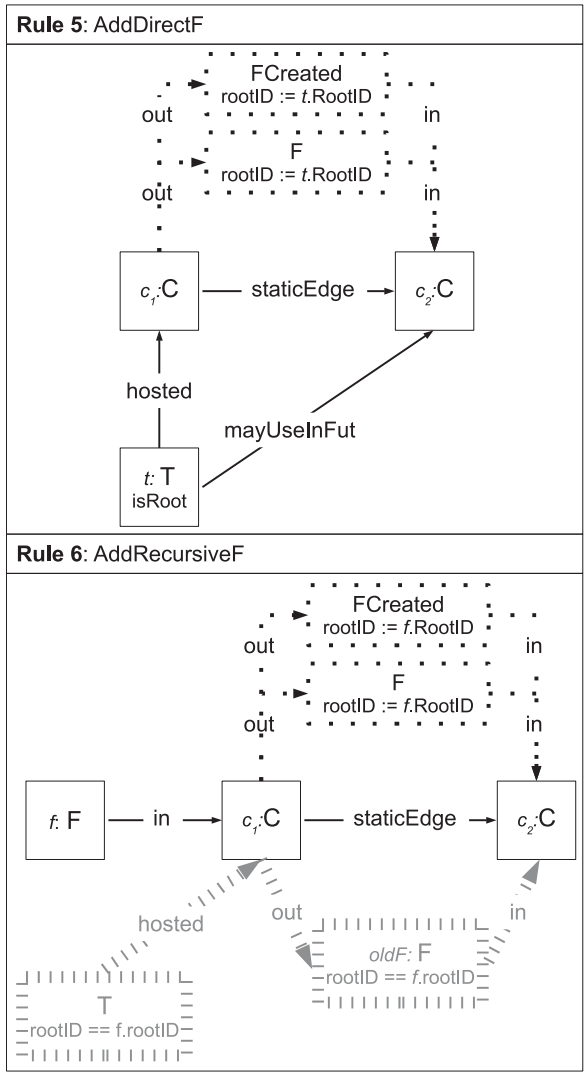


Fig. 7. GT rules for **Setup**.

edges. **Rule 9** models the removal of a future edge from c_1 to c_2, which occurs when a component c_1 will not use the other component c_2 in the context of the same distributed transaction anymore (absence of *mayUseInFut* edge), and there is no future edge, associated with the same transaction, that enters c_1.

Note that, even if the rules show different components, they all model a local decision of component c_1, its currently hosted transaction, and its $f(T)$ (represented by means of *mayUseInFut* edges).

After explaining how to render the management of *future/past* edges, we can define a special-purpose rule (**Rule 10** of Fig. 9) that checks for the freeness of a component c with respect to a transaction T and updates it through the addition (+) of flag updated. The rule is applicable, and thus the component is free, if the component is not hosting any transaction T and there is no pair of f/p edges with the same identifier of root transactions.

The formalization of the clean up step, which we simply omit for brevity, consists of a set of rules that remove the remaining future and past edges after the termination of a root transaction.

The graph transformation rules presented in this section refer to strategy WF. Strategies CV and BF can be reduced to WF by dividing the sub-transactions running on the

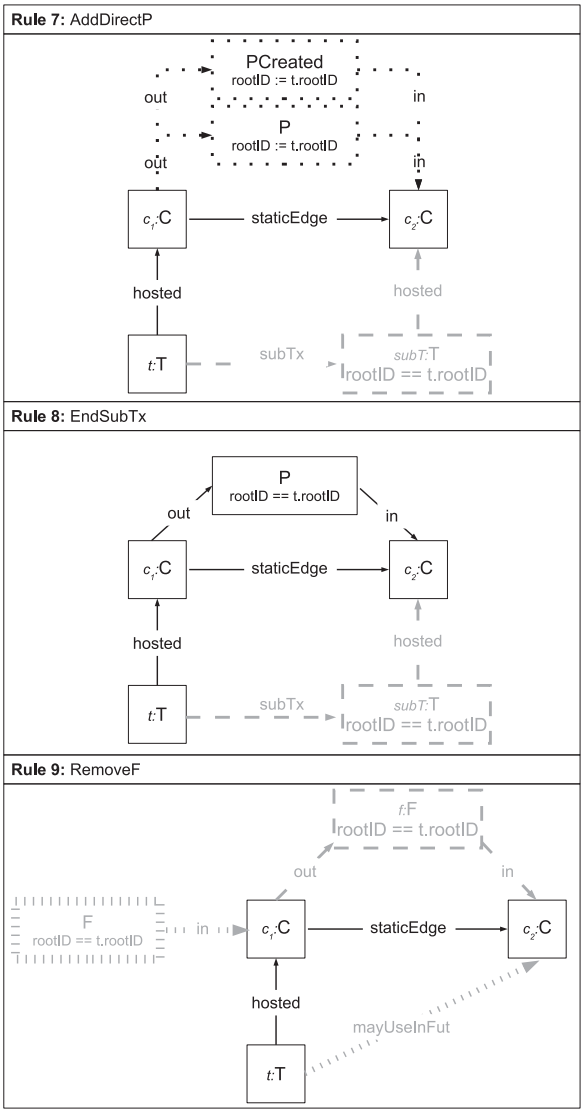


Fig. 8. GT rules for **Progress**.

component-to-update in two groups: those that belong to the extended transaction sets (i.e., distributed transactions) the component has already served or is serving, and those that do not. As for the freeness of a component, the difference between strategy WF and strategies CV and BF is then that in the former case, we consider both sets of sub-transactions, while in the latter we only consider the first one.[6]

6.1 Correctness of Algorithm

We used the graph transformation rules to verify that the application of the proposed algorithm for managing dynamic dependencies satisfies the constraints of a valid configuration and that it finally guarantees version consistency.

If we go back to the definitions of Section 4.1, *locality* is imposed by construction. Since there is no rule that adds f/p edges without a corresponding static edge, and no rule that removes static edges, the property can never be violated. We verified the other properties (future-validity, past-validity, and version consistency) by negation. We used GROOVE to model the graph transformation rules that represent the negation of the properties of interest, and we fed the analyzer with the following CTL formulae: $AG(!ruleViolatedProperty)$, where *ruleViolatedProperty* is the actual rule that negates the property of interest.

Fig. 10 shows the graph transformation rules that negate future validity. The property is divided in two parts, one

6. The rules that formalize strategies CV and BF are part of our complete formalization available at https://github.com/brickinwall/conup

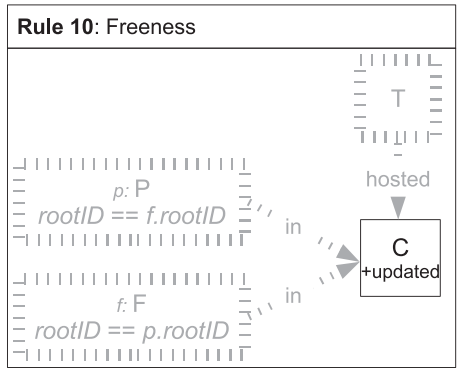


Fig. 9. Update with freeness as GT rule.

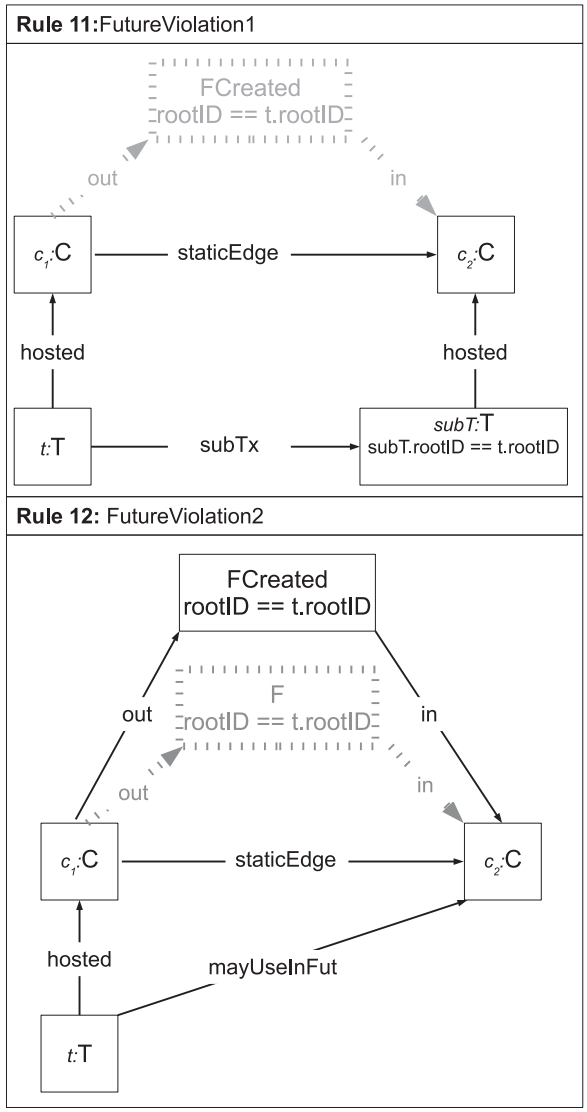


Fig. 10. GT rules that negate future validity.

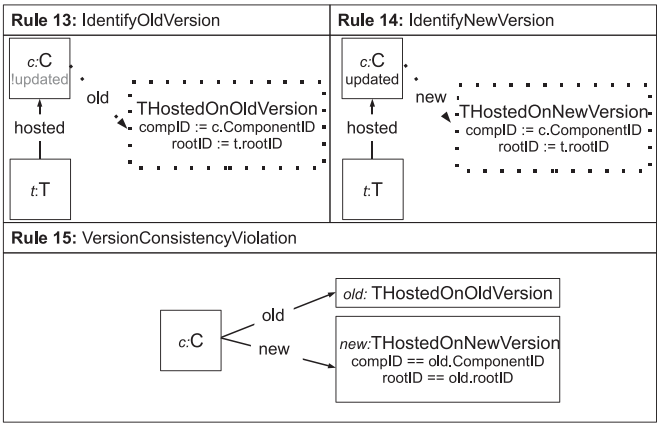


Fig. 11. GT rules that negate version consistency.

describing when a future edge is added and the other describing when it is removed, respectively. **Rule 11** negates the first part of the property and it is applicable if there exists a sub-transaction without any corresponding future edge. The second part of the property is negated by **Rule 12** and it can be applied if there exists a component hosting a transaction that may use the destination component in the future without a corresponding future edge, but with the *FCreated* marker in place. Future validity is verified if both rules are never applicable. Similar rules (omitted here) have been defined to model past validity.

In addition to checking configuration validity, we also analyzed how the dependency management algorithm ensures version consistency. Fig. 11 shows the graph transformation rules used for the verification. **Rule 13** and **Rule 14** create two auxiliary nodes to identify the version of the component that hosts a given transaction. Flag updated is used to identify the most recent version of the component and is added when a component is updated (see **Rule 10**). The absence of this flag (i.e., !updated) is used to identify the version of a component before the update. Version consistency is violated if transactions that belong to the same distributed transaction are hosted on different versions of the same component. This violation is modeled by **Rule 15**, which is only applicable if both the old and new versions have hosted at least one of the transactions that belong to a given distributed transaction (identified by rootID).

The verification of the three properties (i.e., future validity, past validity, and version consistency) has been performed by running the model checker provided by GROOVE to exhaustively search for all possible applications of the graph transformation rules for systems with a fixed number of components, static edges, root transactions, and sub-transactions. Searching for all possible static configurations with a given number n of components is quite complex and the state space (of the model checker) tends to explode even with a small n.

We conducted our experiments on a Mac computer with a 1.7 GHz Intel Core 7 and 8 GB of memory, and we were only able to exhaustively explore static configurations with up to three components and four edges.[7] We then proved the three properties through model checking for the generated configurations.

TABLE 1
State Spaces Explored for All Possible Static Configurations with Three Components and Four Static Edges

# root txs	# sub-txs	State Space (MB)	Exploration Time (s)	# explored states	# explored transitions
1	1	0.645	1.3	2,141	3,689
1	2	1.8	2.4	5,291	10,355
1	3	3.9	4.1	10,397	22,547
2	1	27.7	19.5	62,759	161,417
2	2	123.6	86.8	249,857	715,709
2	3	423.8	341.5	769,307	2,458,043

The results we obtained are shown in Table 1, where each line corresponds to the same system with three nodes and four edges, but with a different number of concurrently executing transactions and sub-transactions. Besides stating the size of the state spaces generated by the model checker (i.e., the memory used by the model checker) and the time needed to conduct the analysis, we also counted the number of states generated by the model checker, that is, the number of different host graphs, and the transitions between them, that is, the number of applied rules. Obtained results say that (i) the algorithm satisfies the properties in the presence of concurrently executing root transactions; (ii) the algorithm satisfies the properties even in the presence of loops in the static configuration. Even though we could not explore large models, due to intrinsic limitations of the verification engine, the *small scope hypothesis* [22] reinforced our confidence in the correctness of the proposed approach.

We also tried to verify the properties on larger systems in a different, less general, way. We generated 100 random static configurations with 100 components, and 101 edges, each. For each configuration, we then activated one single root transaction with 100 sub-transactions running concurrently and randomly. We also decided that when a component was free, it had to be updated to a newer version. We then let each system evolve, that is, we had a random linear exploration of the (possible) state space, and we verified the three properties. Each exploration represents one possible behavior of the system and one corresponding evolution of the algorithm. We were always able to prove the properties, and each exploration took on average 50 ms and 58 KB of memory. We repeated the exploration 100 times to prove the properties for 100 possible evolutions of the algorithm.

6.2 VCC: Version Consistency Checking

The graph transformation rules and GROOVE have been embedded in Version Consistency Checking,[8] a tool that helps engineers select the approach that is more suitable for their dynamic updates. **VCC** allows engineers to model the business logic of each component through state machines. Special-purpose graph transformation rules combine these separate models and materialize all the possible behaviors by matching outgoing and incoming method invocations and replies.

7. Given three nodes, one needs four edges to create at least one loop in the static configuration.

8. The interested reader can refer to https://github.com/brickinwall/conup for a detailed video presentation of the tool.

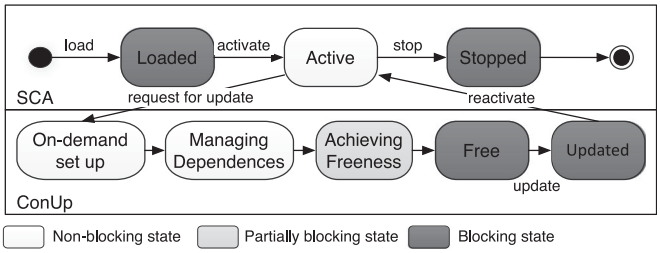


Fig. 12. Lifecycle of ConUp components.

VCC then uses another graph transformation rule to automatically check whether the application of the tranquility-based approach violates version consistency, that is, the safe update of the system. If version consistency is violated, the tool provides a counter example that shows when and how the violation occurs.

7 ConUp

ConUp is the prototype framework we developed to implement our approach. It is a component-based framework that allows for the safe, timely, and low-disruptive dynamic updates of components on top of the Service Component Architecture, [10] component model.

SCA is a lightweight component model, where components are specified in terms of the services they offer and the services they require. The resulting model can then easily be mapped onto the static configuration required by our approach. The specification is completely decoupled from the implementation and favors interoperability. Components can be implemented in a wide set of programming languages—such as Java, C++, and PHP—frameworks, and environments—such as BPEL, EJB and Spring. SCA also allows components to communicate through different standards, including SOAP, JMS and RPC. This decoupling eases the porting of existing systems on top of SCA, and thus fosters the dynamic update of legacy systems that were not conceived for run-time maintenance.

ConUp extends SCA with the following capabilities:

- **Transaction management** handles the lifecycle of transactions and helps propagate the identifiers of root transactions and the notifications of creation and completion of sub-transactions.
- **Dependency management** maintains a runtime model of the system, which reifies the dynamic dependencies between components, and helps understand when a component can be safely updated.
- **Component lifecycle management** improves native management with (i) a versioning mechanism that distinguishes among the different versions of a component at runtime, (ii) a finer-grained component lifecycle model (Fig. 12), and (iii) a management interface for managing update requests at runtime.

The lifecycle model of Fig. 12 extends the common lifecycle of SCA components with the states needed for the management of dynamic updates and highlights how the management of dependencies is key for the dynamic update of components. State *On-demand set up* refers to the on-demand set up of dynamic dependencies. Note that the configuration is not valid in this state yet, but it is always valid in state *Managing Dependencies*. State *Achieving Freeness* is actually a composite state whose internals depend on the actual strategy used for achieving freeness.

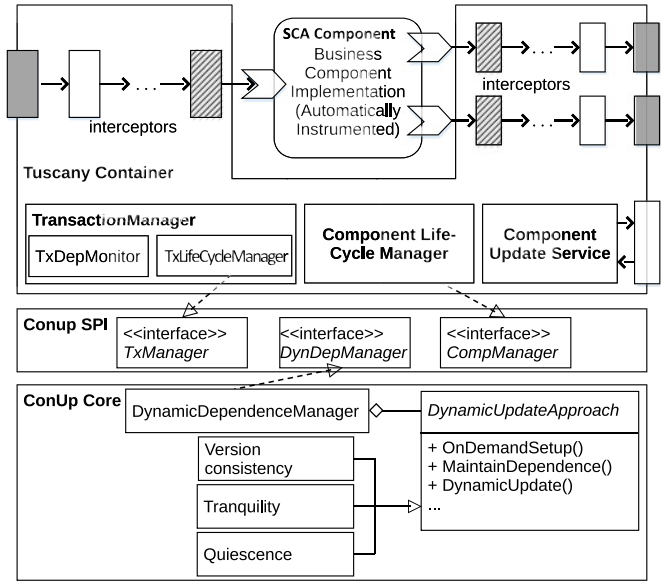


Fig. 13. ConUp architecture.

7.1 Design and Implementation of ConUp

ConUp extends Apache Tuscany,[9] a well-known lightweight implementation of the SCA specification, but it is designed to easily be ported onto other component frameworks, like JBoss or Tomcat. Moreover, ConUp supports quiescence, tranquility,[10] and version consistency, as possible approaches for updating components at runtime, and BF, CV, and WF as strategies for achieving freeness (Section 5.1).

Fig. 13 presents the three-layered architecture of ConUp. At the bottom, layer **ConUpCore** implements the actual dynamic dependency management and is responsible for making the component become free of dependencies and thus for enabling its safe update. The modules in this layer are also responsible for the adoption of the different approaches and strategies.

ConUp Core cannot accomplish its tasks without the help of the component framework. Layer **Tuscany Container** augments the standard Tuscany runtime with module *Transaction Manager* for the creation, management, and monitoring of local transactions. This module also takes care of sets $f(T)$ and $p(T)$ for dependency management. It also adds module *Component Lifecycle Manager*, which supports the lifecycle model of Fig. 12 and is responsible for the deployment of new versions of components and for the undeployment of old ones.

When a component needs to be updated, the user (i.e., the system administrator) initiates the update process by invoking *Component Update Service* with the indication of the new version of the component-to-update.

To improve the reusability of ConUp core, we introduced layer **ConUpSPI** between the two layers above. It reifies the

9. http://tuscany.apache.org/

10. ConUp implements tranquility by simply restricting the consistency scope of each transaction to the transaction itself and its direct sub-transactions.

aforementioned extensions to SCA as a set of *service provider interfaces*. In principle any component framework that implements these interfaces can use ConUp to perform dynamic updates.

7.2 Declaration of Transactions

The algorithm described in Section 5 assumes that the host component of each transaction T knows $f(T)$ and $p(T)$. While $p(T)$ can easily be recorded when sub-transactions terminate, $f(T)$ must properly be computed at runtime. Our approach is flexible and $f(T)$ can be conservatively estimated. If no information about dynamic dependencies is available, our approach degenerates into something similar to the quiescence-based one.

ConUp requires that (local) transactions be explicitly declared. Currently ConUp supports Java as implementation language for SCA components and supplies a special-purpose annotation `@ConupTx` to mark the public methods that correspond to transactions. However, one can also implement a mechanism that exploits external configuration files to designate transaction boundaries automatically. Such a mechanism would be completely non-intrusive and also applicable to legacy systems.

We have then developed a fully automated solution for the estimation of $f(T)$ based on basic control-flow analysis. Given a particular node n of the control flow graph of transaction T on component c_1, the static edge se, which connects c_1 and c_2, is kept in $f(T)$, iff c_2 is reachable from n, that is, T will use c_2 in the future. The implementation of this idea is based on byte-code level control-flow analysis and load-time instrumentation, and the runtime overhead is well controlled. Interested readers can refer to [23] for a detailed presentation.

All the experiments presented in the next section exploit this solution. Obtained results indicate that even this straightforward estimation is sufficient to witness significant efficiency improvements over the quiescence-based solution.

8 Evaluation

This section summarizes the experiments we conducted to evaluate the timeliness and degree of disruption of our approach and of its companion implementation ConUp.

In previous work [4], we used simulation to evaluate the timeliness and disruption of our approach for a wide set of randomly generated CBDSs that varied in the number of components, service time, and network latency. The results show that dynamic updates based on version consistency are on average 20 percent more timely and 50 percent less disruptive than those based on quiescence. In this paper, we use *travel sample*,[11] an existing SCA-based application to show how our approach and ConUp can be effective in practice. We also applied the approach based on tranquillity on this application, but found that, unlike the approaches based on quiescence and version consistency, it cannot always guarantee consistency: the possibility of being inconsistent varies between 5 and 10 percent and depends on the workload. This is why we decided not to consider this approach in our evaluation. On the other hand, when the tranquility-based solution is known to be sufficient, one should use it, maybe through ConUp.

11. The original implementation of the application can be found at: http://tuscany.apache.org/sca-java-travel-sample-1x-releases.html

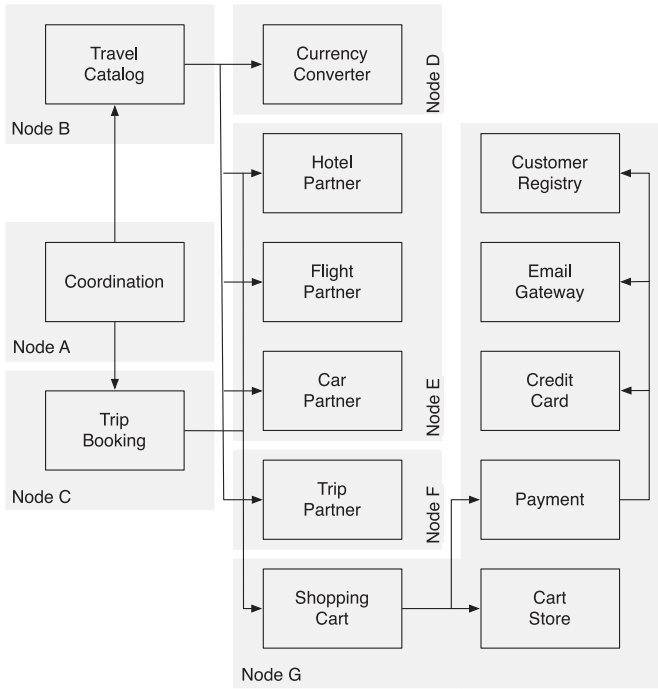


Fig. 14. Travel sample: Static configuration.

8.1 Travel Sample

This application is a travel booking web system released with the Tuscany runtime. We extended the original application and tested the dynamic update of a few components. We used ConUp to apply version consistency-, quiescence-, and tranquility-based updates.

Fig. 14 shows the static configuration of the system, which consists of fourteen SCA components. Users send requests to Coordination to search for travel offers, to book the desired travel combination, to add it to the cart, and to conclude with the payment. A travel offer is constructed as a combination of different third-party services managed by separate components: HotelPartner, FlightPartner, CarPartner, and TripPartner are in charge of interacting with the third-party services devoted to deal with hotels, flights, cars and trips, respectively. Component TravelCatalog returns a list of available combinations of the different services and interacts with CurrencyConverter if the resulting offers require currency conversions. Component TripBooking manages the booking of the desired offer and interacts with ShoppingCart and with the single providers for confirmation. When the travel offer is booked, ShoppingCart adds it to the CartStore and proceeds with Payment (the scenario is shown in Fig. 15).

A number of possible updates may happen at run-time: (i) HotelPartner, FlightPartner, CarPartner, and TripPartner may be dynamically reconfigured to substitute the services they offer or to include new ones; (ii) CurrencyConverter could be changed to support more currencies or to change the frequency for checking the conversion rates; (iii) Payment and CreditCard could be changed to introduce new security countermeasures or to support different payment services. Even with these simple scenarios, improper

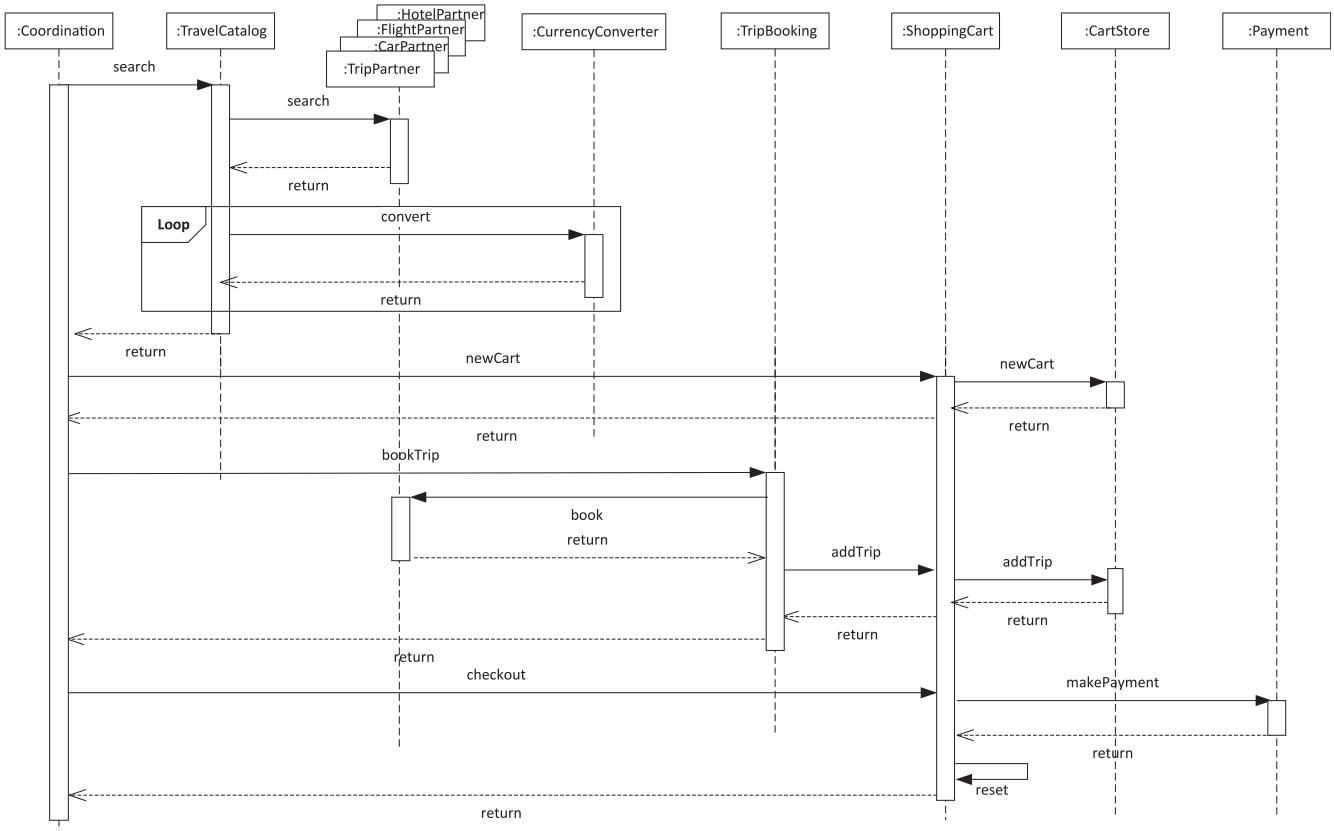


Fig. 15. Travel sample: Scenario.

dynamic updates may lead to inconsistencies. As an example, after finding a car to rent, this would not be available anymore if the CarPartner were updated, or if the currency conversion were not valid anymore before the actual payment.

8.2 Experimental Results

The experiments we carried out were based on the following setting. The 14 components of the application were deployed on seven nodes as shown in Fig. 14. Each node was a virtual machine configured with a dual-core 2.66 GHz CPU and 4 GB of memory, and the network was a virtualized Gigabit ethernet. The underlying software environment was Ubuntu 12.04 (64 bit) with JDK 1.7.0_05.

Our first goal was a comparison between quiescence- and version consistency-based updates in terms of timeliness and disruption. We implemented quiescence on top of ConUp by recursively passivating all the components that statically depend on the ones-to-update. ConUp can also simulate tranquility by restricting the scope of consistency for each transaction to itself and its direct sub-transactions. However, the tranquility-based solution is not included in the comparison because the example application is beyond its scope. For example, the use of a tranquillity-based solution for the dynamic update of TripPartner would allow one to book trips that do not exist anymore or that have become more expensive.

We ran the application with different workloads by generating requests to Coordination. Requests arrived according to a Poisson process whose inter-arrival times satisfy an exponential distribution. To simulate the typical delays associated with the invocation of external services, we introduced a 200 ms delay for the *search* methods of HotelPartner, FlightParter, CarPartner and TripPartner, and a 2,000 ms delay for method *makePaymentMember* of Payment.[12]

The *timeliness* of dynamic updates is measured as the timespan between receiving a request for update and its actual completion. If the dynamic update of a component happens while a transaction is running, that transaction is said to be *affected* by the update. The *total disruption* is then the sum of the delays caused by affected transactions with respect to the response time of the same transactions without update. Finally, the *average disruption* is the total disruption divided by the number of affected transactions.

The results presented in this paper are for the update of two representative components, CurrencyConverter and TripPartner. Fig. 15 shows that component CurrencyConverter was only used once (although within a loop) in the same distributed transaction, and the prediction it would have not been used anymore can be done at a very early stage because CurrencyConverter is only used by TravelCatalog, which is only used by Coordination, and Coordination finishes its use of TravelCatalog early. Thus the dynamic dependency between TravelCatalog and CurrencyConverter only held for a limited amount of time with respect to the total duration of the distributed transaction. This is why, intuitively, our approach was significantly more efficient than the quiescence-based one in this case.

12. We use fixed values instead of randomized delays to ease the measurement of the disruption caused by dynamic updates.

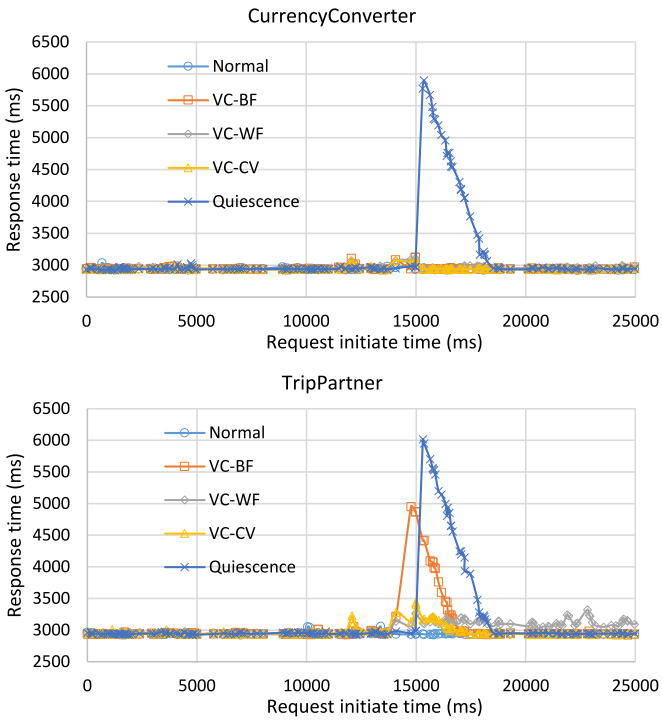


Fig. 16. Response times.

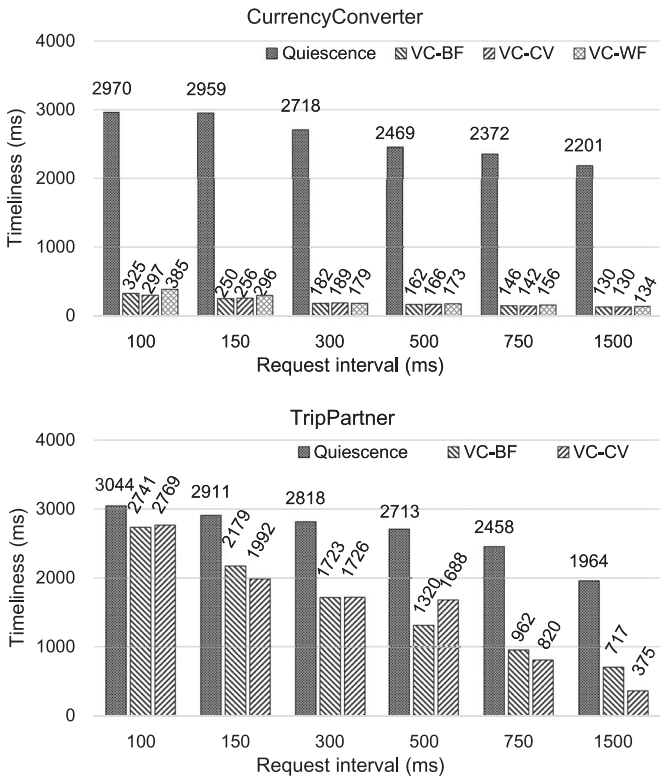


Fig. 17. Timeliness.

TripPartner, instead, was used multiple times by different components and the prediction had to be very conservative because of the iterative logic of TripBooking. This is more challenging for our approach.

The experiments also aimed to analyze the impact of the different strategies for achieving freeness on the timeliness and disruption of the actual update. Especially, we wanted to see when WF (simply based on waiting) can be applied and when we must use CV(concurrent versions) or BF (blocking components).

8.2.1 *Disruption and Timeliness*

Fig. 16 plots the response times of root transactions continuously initiated before, during, and after the dynamic update of a component. Normal refers to the execution of transactions without any attempt to update components and provides the baseline for evaluation. These experiments—-one for CurrencyConverter and one for TripPartner—issued requests at a rate following an exponential distribution with a mean request interval of 150 ms. The request for update was issued exactly at time 15 s, which resulted in some significant delays in some cases (the spikes in the figure). Note that the *x*-axes identify the time points when the corresponding requests were *initiated*, not when responded. This means that a request initiated at time 14 s could be served by the system at time 15 s, then it could be affected by the dynamic update and its response be higher than usual. This higher value would be plotted at time 14 s although the delay actually happened after time 15 s.

When updating CurrencyConverter, the quiescence-based solution caused a delay of up to 3 s to the response time of affected transactions, while version consistency only introduced a negligible disruption, no matter the strategy used for achieving freeness. When updating TripPartner, the quiescence-based solution behaved similarly as above, but version consistency exhibited different delays with respect to the different strategies for achieving freeness. BF caused delays of up to 2 s to affected transactions, while CV was much better as it only caused delays shorter than 500 ms. WF could not achieve freeness within 30 s, thus no update happened, and the only registered overhead was the one consumed to maintain the dynamic dependencies.

These diagrams intuitively show that version consistency performs better than the quiescence-based solution, and the gain is more significant when CV is used. It also shows that the overhead introduced by version consistency is negligible before and after the update. This is due to the fact that the dynamic dependencies are set on-demand and they are only maintained while the component is updated.

We then conducted additional experiments with a mean inter-arrival time between 100 ms and 1,500 ms. Fig. 17 shows the timeliness of the aforementioned dynamic updates under different levels of workload. The data are averages over 10 runs. Again, when updating CurrencyConverter, version consistency is much more timely than the quiescence-based solution, but its advantage is less significant when updating TripPartner. Note that in the latter case, the adoption of WF makes the update not finish within 30 s for root transaction request intervals shorter than 500 ms. This is why we decided to omit this series of columns from the diagrams that refer to the update of TripPartner with WF.

This result is explained by the high sensitivity of WF for high workloads. More running transactions mean fewer opportunities for a spontaneous achievement of freeness. Given these results, we recommend that CV be adopted; BF could be a good alternative if the concurrent

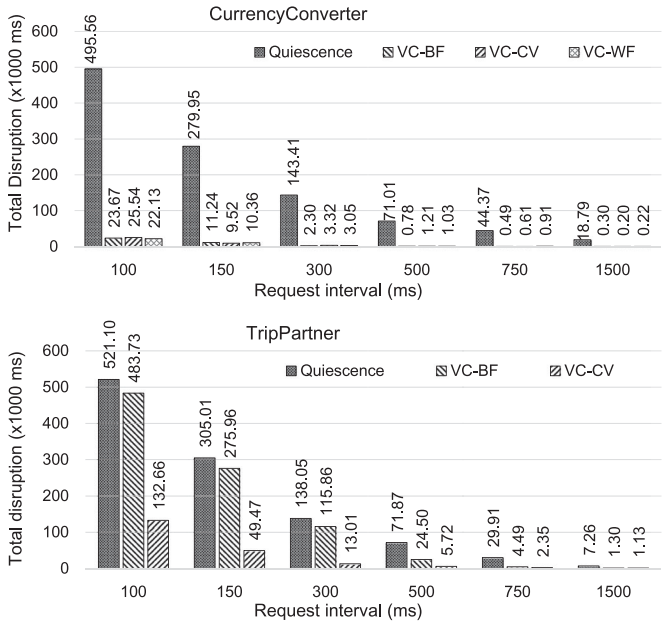


Fig. 18. Total disruption.

presence of multiple versions of the same component is not feasible. The adoption of WF is only suggested when the workload is light.

Finally, Figs. 18 and 19 show the total and average disruptions, respectively. Also in this case, the data are averages over 10 runs. The results show that CV, BF, and WF considerably reduce the average level of disruption with respect to the quiescence-based solution for the dynamic update of both CurrencyConverter and TripPartner. Among the three proposed strategies, BF is more disruptive since new transactions are blocked to ensure freeness. When CV or WF are used, our approach only introduces negligible disruption, which is due to the overhead of the on-demand set up and to the maintenance of configurations.

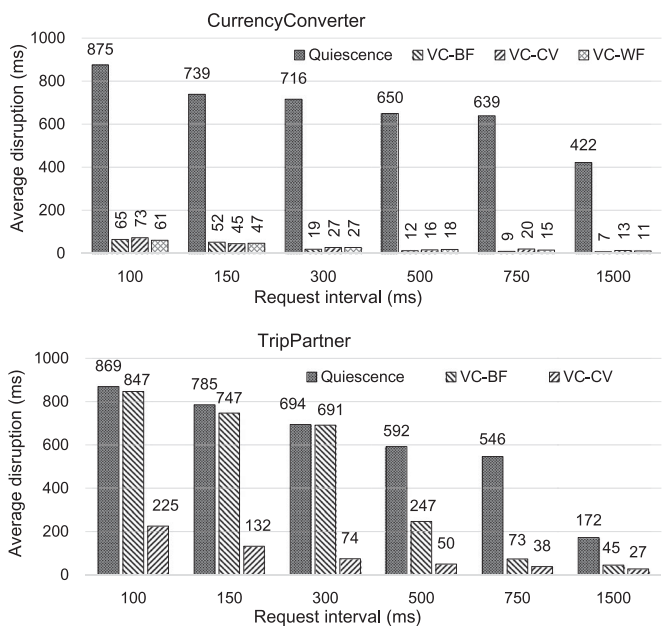


Fig. 19. Average disruption.

TABLE 2
Overhead Imposed by ConUp

Req. Intvl. (ms)	1,500	750	500	300	150	100
Tuscany (ms)	2,943	2,929	2,933	2,940	2,941	2,944
ConUp (ms)	2,964	2,977	2,966	2,974	2,970	2,969
Overhead (%)	0.71	1.67	1.12	1.14	0.98	0.85

8.2.2 Costs of Our Approach

The advantages of our approach do not come for free. Compared to "pure" Tuscany, ConUp pays extra costs for: (1) analyzing and instrumenting the byte code of components at load-time; (2) executing the code generated by the analysis at runtime; (3) managing the lifecycle of transactions and propagating root transaction ids among transactions; (4) setting up dynamic dependencies on-demand; (5) managing dynamic dependencies after on-demand setup and before update; and (6) achieving freeness through strategies BF, CV or WF.

The first cost is small and one-off. For the example application, the time for the analysis and instrumentation of each component varies from 20 to 80 ms.

To evaluate the second and third costs, we ran the travel sample application on "pure" Tuscany and on ConUp respectively. Table 2 shows the response times of the application running on the two platforms with different request intervals (Req. Intvl). The data are averages of 200 root transactions. The result indicates that the overhead is only 1-2 percent, and thus is negligible.

The remaining costs have already been described in terms of disruption in Section 8.2.1. Here we further analyze them in terms of number of management messages used to ensure the consistency of dynamic updates.

The quiescence-based approach only needs 6 and 12 messages to update CurrencyConverter and TripPartner, respectively. Version consistency may need many more messages to setup and manage dynamic dependencies. Table 3 lists the number of affected root transactions (Tx) and the corresponding number of required messages (Msgs) using strategies BF, WF and CV, respectively. The data are averages over 10 runs. 'N/A' in the table means that freeness was not achieved within 30 s.

Since in our example application each root transaction causes the execution of 48 sub-transactions, and each of them requires at least two messages, our approach only

TABLE 3
Management Messages

Req. Intvl. (ms)		1,500	750	500	300	150	100
Currency Converter	#Tx BF	2.1	3.9	8.2	9.5	22.4	33.7
	#Msgs BF	9.3	15.0	23.6	28.2	56.3	90.6
	#Tx WF	1.7	3.2	5.2	11.1	21.5	30.7
	#Msgs WF	8.0	13.0	16.9	35.1	62.6	84.4
	#Tx CV	2.0	4.5	5.8	11.5	24.4	29.4
	#Msgs CV	8.6	14.0	18.9	34.7	68.9	90.2
Trip Partner	#Tx BF	2.8	6.0	7.7	16.0	35.6	55.6
	#Msgs BF	33.1	50.5	59.5	105.1	250.9	372.8
	#Tx WF	2.8	4.1	N/A	N/A	N/A	N/A
	#Msgs WF	19.2	30.2	N/A	N/A	N/A	N/A
	#Tx CV	1.9	4.3	7.6	15.4	33.5	58.9
	#Msgs CV	19.2	40.1	92.2	200.8	384.7	745.6

adds a small fraction (i.e., $\frac{\#\text{Msgs}}{\#\text{Tx}_{\times 48 \times 2}}$, which is less than 13.6 percent) to the total number of messages exchanged during a dynamic update.

However, we also observed rare cases in which actual costs are higher and vanish the benefits of our approach. This happens when many components depend on the same component, the workload is high, and strategy BF is used. For example, if we considered HotelPartner, and request intervals of 100 and 150 ms, our approach, with strategy BF, performs 20-30 percent worse than the quiescence-based one in terms of average disruption. Compared with the result of TripPartner shown in Fig. 19, one may conclude that this is due to the only essential difference between the two components: as shown in the configuration of Fig. 14, HotelPartner is deployed together with two other similar components on node E, and thus it becomes more sensitive to high workloads than TripPartner, which is alone on node F. In contrast, the use of strategy CV would outperform the quiescence-based solution.

9 Related Work

The dynamic update of running applications has been extensively studied in different areas like programming languages [11], [24], [25], [26], operating systems [27], [28], [29], and software engineering [30], [31]. A common theme of these works is the selection of proper time points when the state of the system is steady and ready for applying a user-specified state transformation. The result is a new valid state from which the system is able to continue its execution. Since generally the validity of the resulting state is undecidable [11], most research efforts focus on: (a) human-assisted identification of proper time points and state transformers, by providing necessary analysis tools and runtime libraries, and (b) improving the timeliness of updates by automatically deriving further safe time points from those specified by the user.

Instead of switching from the old to the new version of the application, some researchers proposed intermediate versions to smooth the adaptation process. For example, Zhang and Cheng [2] propose a model-based approach for the development of adaptive software. The behaviors of the different versions of an application are modeled by different state machines, and the adaptation behavior is rendered through states/transitions that connect them. Biyani and Kulkarni [32] use adaptation lattices to model transition paths from old to new programs, and introduce the concept of transitional-invariant lattice to verify the correctness of adaptation. Boyer et al. [33] propose a reconfiguration protocol that incrementally modifies the architecture of the system towards the new target one and respects architecture invariants at each step of the process.

As for CBDSs, it is possible to avoid the direct manipulation of application-specific states of components and to maintain a clear separation of concerns between reconfiguration management and application logic. The focus is often on identifying suitable conditions (abstract states) under which components can be safely manipulated. The concept of quiescence by Kramer and Magee [1] is a prominent early work, but it may impose too high disruption on system service. Subsequently, the dynamic reconfiguration service for CORBA by Bidan et al. [5], the proposal by Chen [6], and the idea of tranquillity by Vandewoude et al. [3] reduce disruption by considering dynamic dependencies, but they only guarantee some local consistency properties or impose stringent restrictions on the systems on which the approaches can be applied.

Wojciechowski et al. [34], [35] study dynamic protocol update (DPU) for distributed systems. In DPU the subject to be updated at runtime is a *horizontal* layer of a protocol stack that is installed on every node of the system. The goal is to ensure a correct order of message delivery despite the dynamic update. This work is complementary to ours in the sense that we consider the update of a single entire node (component), which is a *vertical* slice of the system.

It is also widely recognized that the dynamic adaptation of software systems should be modeled, analyzed, and managed at architectural level [36], [37]. Architectural models provide abstract global views of systems and explicitly specify system-level integrity constraints that must be preserved by reconfiguration. However, these models do not usually provide information about the dynamic dependencies needed to perform safe and low-disruptive runtime reconfigurations. There are few formal models that cover both local computations and architectural reconfigurations. For example, Wermelinger et al. [38] propose a category theory-based approach for the uniform modeling of the computations performed by components and their architectural configurations, and Taentzer et al. [18] propose the use of distributed graph transformation systems to model configurable distributed systems. Compared to these proposals, our approach strikes a better balance in separating dynamic reconfiguration from computation and allows for safe but efficient dynamic reconfigurations. This is due to the use of future and past edges as simple (but powerful) abstraction of the dynamic dependencies between components.

As the name suggests, our version consistency criterion is inspired by the work on transactional version consistency by Neamtiu et al. [39]. This work focuses on the dynamic update of centralized programs at code level, and its notion of transaction is a user-specified scope in the code. Their approach ensures that the execution of the code in the scope comply with the same, single version, no matter when the update happens. This approach relies on static program analysis and cannot be used in a distributed setting.

Modern application servers such as Tomcat and JBoss often provide *hot deployment* capabilities. A component can be updated to a new version without restarting the server. Especially, FraSCAti [40] is a recent SCA implementation with enhanced dynamic reconfiguration support and runtime management features. However, these servers do not manage the dynamic dependencies between components and thus cannot ensure the global consistency of applications in the presence of dynamic updates.

Canavera et al. [41] propose an interesting approach to obtain dependencies between components by mining system execution logs. This approach can be used when the logs are available. Differently from this approach, ConUp derives dependencies automatically from the byte code of components and always ensures consistency.

10 Conclusions

The dynamic update of system components is widely desired, but its practical application is still limited. This is partly because of the complexity of balancing the consistency of changes with the disruption of provided service. The use of version consistency as criterion for the safe dynamic update of distributed component-based systems aims to alleviate the problem. The approach does not compromise the correctness of distributed transactions, but it allows for better timeliness and lower disruption than previous approaches. ConUp provides the prototypical component framework to apply version consistency on real systems, and to conduct experiments with other approaches. An evaluation of the proposed approach witnesses its merits. The evaluation presented in the paper only compares our approach against the one based on quiescence. We do not report the results we obtained with the tranquility-based approach since, in our example, it is not able to guarantee consistent updates in some 5-10 percent of the cases we analyzed.

We believe that the approach proposed in this paper can be further extended. For example, the current solution uses a pessimistic tactic for avoiding inconsistency, but a more optimistic one could be more efficient and lightweight, especially when the underlying middleware platform already provides some recovery mechanisms, like rollback and compensation capabilities. These optimizations, along with the extension to the update of sets of components atomically will be the target of our future work.

Acknowledgments

We want to thank Guochao Ren, Jiang Wang, and Yiqun Wang for their contribution to the implementation of ConUp. The work presented in this paper has been partially supported by the 973 Program of China under grant No. 2015CB352202, NSFC under grants No. 61472177 and 91318301, by project EEB-Edifici a zero consumo energetico in distretti urbani intelligenti (Italian Technology Cluster For Smart Communities)-CTN01_00034_594053, and by Telecom Italia, which supported Valerio Panzica La Manna while he was a postdoctoral researcher at the Politecnico. The work presented in this paper was carried out while Valerio was at the Politecnico di Milano, Italy. Xiaoxing Ma is the corresponding author of this paper.

References

[1] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Trans. Softw. Eng.*, vol. 16, no. 11, pp. 1293–1306, Nov. 1990.

[2] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 371–380.

[3] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 856–868, Dec. 2007.

[4] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu, "Version-consistent dynamic reconfiguration of component-based distributed systems," in *Proc. 8th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2011, pp. 245–255.

[5] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras, "A dynamic reconfiguration service for CORBA," in *Proc. 4th Int. Conf. Configurable Distrib. Syst.*, 1998, pp. 35–42.

[6] X. Chen and M. Simons, "A component framework for dynamic reconfiguration of distributed systems," in *Proc. IFIP/ACM Work. Conf. Compon. Deployment*, 2002, pp. 82–96.

[7] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*, 1st ed. San Mateo, CA, USA: Morgan Kaufmann, Sep. 1992.

[8] L. Baresi and R. Heckel, "Tutorial introduction to graph transformation: A software engineering perspective," in *Proc. 1st Int. Conf. Graph Transformation*, 2002, pp. 402–429.

[9] A. Ghamarian, M. Mol, A. Rensink, E. Zambon, and M. Zimakova, "Modelling and analysis using GROOVE," *Int. J. Softw. Tools Technol. Transfer*, vol. 14, no. 1, pp. 15–40, 2012.

[10] Service Component Architecture (SCA) | Oasis OpenCSA. (2011). [Online]. Available: http://www.oasis-opencsa.org/sca

[11] D. Gupta, P. Jalote, and G. Barua, "A formal framework for online software version change," *IEEE Trans. Softw. Eng.*, vol. 22, no. 2, pp. 120–131, Feb. 1996.

[12] M. Vatkina, Ed., *JSR 345: Enterprise JavaBeans, Version 3.2*. Redwood City, CA, USA: Oracle, 2013.

[13] M. Papazoglou, *Web Services and SOA: Principles and Technology*, 2nd ed. London, U.K.: Pearson, 2012.

[14] J. E. Cook and J. A. Dage, "Highly reliable upgrading of components," in *Proc. 21st Int. Conf. Softw. Eng.*, 1999, pp. 203–212.

[15] S. Ajmani, B. Liskov, and L. Shrira, "Modular software upgrades for distributed systems," in *Proc. 20th Eur. Conf. Object-Oriented Programming*, 2006, pp. 452–476.

[16] D. P. Mitchell and M. J. Merritt, "A distributed algorithm for deadlock detection and resolution," in *Proc. 3rd Annu. ACM Symp. Principles Distrib. Comput.*, 1984, pp. 282–284.

[17] D. Le Metayer, "Describing software architecture styles using graph grammars," *IEEE Trans. Softw. Eng.*, vol. 24, no. 7, pp. 521–533, Jul. 1998.

[18] G. Taentzer, M. Goedicke, and T. Meyer, "Dynamic change management by distributed graph transformation: Towards configurable distributed systems," in *Proc. Sel. Papers 6th Int. Workshop Theory Appl. Graph Transformations*, 2000, pp. 179–193.

[19] R. Bardohl, M. Minas, G. Taentzer, and A. Schürr, "Application of graph transformation to visual languages," in *Handbook of Graph Grammars and Computing by Graph Transformation*, H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds. Singapore: World Scientific, 1999, pp. 105–180.

[20] G. Taentzer, "AGG: A graph transformation environment for modeling and validation of software," in *Proc. 2nd Int. Workshop Appl. Graph Transformations Ind. Relevance*, 2004, pp. 446–453.

[21] L. Baresi and P. Spoletini, "On the use of alloy to analyze graph transformation systems," in *Proc. 3rd Int. Conf. Graph Transformations*, 2006, pp. 306–320.

[22] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA, USA: MIT Press, 2006.

[23] P. Su, C. Cao, X. Ma, and J. Lü, "Automated management of dynamic component dependency for runtime system reconfiguration," in *Proc. 20th Asia-Pacific Softw. Eng. Conf.*, 2013, pp. 450–458.

[24] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Trans. Programming Languages Syst.*, vol. 27, no. 6, pp. 1049–1096, 2005.

[25] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis mutandis: Safe and predictable dynamic software updating," *ACM Trans. Programmming Languages Syst.*, vol. 29, no. 4, Aug. 2007, Art. no. 22.

[26] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: A VM-centric approach," in *Proc. 30th ACM SIGPLAN Conf. Programming Language Des. Implementation*, 2009, pp. 1–12.

[27] A. Baumann, et al., "Providing dynamic update in an operating system," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2005, pp. 32–32.

[28] K. Makris and K. D. Ryu, "Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 327–340.

[29] C. Giuffrida and A. S. Tanenbaum, "Cooperative update: A new model for dependable live update," in *Proc. 2nd Int. Workshop Hot Topics Softw. Upgrades*, 2009, pp. 1–6.

[30] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "POLUS: A powerful live updating system," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 271–281.

[31] S. C. Previtali, "Dynamic updates: Another middleware service?" in *Proc. 1st Workshop Middleware-Appl. Interaction*, 2007, pp. 49–54.

[32] K. N. Biyani and S. S. Kulkarni, "Assurance of dynamic adaptation in distributed systems," *J. Parallel Distrib. Comput.*, vol. 68, no. 8, pp. 1097–1112, 2008.
[33] F. Boyer, O. Gruber, and D. Pous, "Robust reconfigurations of component assemblies," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 13–22.
[34] P. T. Wojciechowski and O. Rütti, "On correctness of dynamic protocol update," in *Proc. 7th IFIP Conf. Formal Methods Open Object-Based Distrib. Syst.*, 2005, pp. 275–289.
[35] O. Rütti, P. T. Wojciechowski, and A. Schiper, "Structural and algorithmic issues of dynamic protocol update," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp.*, 2006, pp. 133–133.
[36] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Comput.*, vol. 37, no. 10, pp. 46–54, Oct. 2004.
[37] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Runtime software adaptation: Framework, approaches, and styles," in *Proc. Companion 30th Int. Conf. Softw. Eng.*, 2008, pp. 899–910.
[38] M. Wermelinger, A. Lopes, and J. L. Fiadeiro, "A graph-based architectural (re)configuration language," in *Proc. 8th Eur. Softw. Eng. Conf. Held Jointly 9th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2001, pp. 21–32.
[39] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis, "Contextual effects for version-consistent dynamic software updating and safe concurrent programming," in *Proc. 35th Annu. ACM SIGPLAN-SIGACT Symp. Principles Programming Languages*, 2008, pp. 37–49.
[40] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A component-based middleware platform for reconfigurable service-oriented architectures," *Softw.: Practice Experience*, vol. 42, no. 5, pp. 559–583, 2012.
[41] K. R. Canavera, N. Esfahani, and S. Malek, "Mining the execution history of a software system to infer the best time for its adaptation," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 18:1–18:11.