A Safe Stopping Protocol to Enable Reliable Reconfiguration for Component-Based Distributed Systems

Mohammad Ghafari, Abbas Heydarnoori, and Hassan Haghighi

DeepSE Group @ Politecnico di Milano, Italy
Sharif University of Technology, Iran
Shahid Beheshti University, Iran
mohammad.ghafari@polimi.it, heydarnoori@sharif.edu, h_haghighi@sub.ac.ir

Abstract. Despite the need for change, highly available software systems cannot be stopped to perform changes because disruption in their services may consequent irrecoverable losses. Current work on runtime evolution are either too disruptive, e.g., "blackouts" in unnecessary components in the *quiescence criterion* approach or presume restrictive assumptions such as the "black-box design" in the *tranquility* approach. In this paper, an architecture-based approach, called *SAFER*, is proposed which provides a better timeliness by relaxing any precondition required to start reconfiguration. We demonstrate the validity of the SAFER through model checking and a realization of the approach on a component model.

Keywords: Reconfiguration, Safe stopping, Consistency.

1 Introduction

Many software-intensive systems are required to be reconfigured to maintain the key functions while they face changes in user requirements and/or domain assumptions. In some special category of software systems, it may not be possible to simply shut down the software from functioning and then apply the changes. In this regard, runtime evolution[1] aims at adapting the system to changes without disrupting those parts of the system which are unaffected by the change [1]. The three most important issues which must be addressed in the runtime evolution are (i) reaching a safe application state, (ii) ensuring reliable reconfiguration, and (iii) transferring the internal state of entities which have to be replaced. Despite extensive research in component-based dynamic reconfiguration, and available component models which allow reconfiguration [2], safe reconfiguration is still an open problem, and existing approaches have made small steps in solving real world scenarios [3].

This paper focuses on the first two challenges of runtime evolution, i.e., reaching a safe application state and ensuring a reliable reconfiguration. It proposes

[1] Interchangeably in this paper, dynamic reconfiguration.

© IFIP International Federation for Information Processing 2015
M. Dastani and M.Sirjani (Eds.): FSEN 2015, LNCS 9392, pp. 100–109, 2015.
DOI: 10.1007/978-3-319-24644-4_7

an architecture-based reconfiguration approach, called *SAFER*[2], that provides a better timeliness by relaxing any preconditions required to start reconfiguration. The approach extends the notion of tranquility in a way that not only enjoys the low disruption of this proposal, but also works safely in both distributed and interleaved transactions. The paper presents the formalization of the proposed approach in Alloy [4] and the verification of its consistency by means of model checking in different architectural configurations and also for the most well known evolution scenarios [5]. Implementation of a running example on top of the Fractal [6] shows the applicability of this approach [5].

This paper is organized as follows. Section 2 gives an overview of the challenges posed by runtime evolution. SAFER, an approach for ensuring safe dynamic reconfiguration, is articulated in Section 3. Related work are then discussed in Section 4, and Section 5 concludes the paper. The verification and evaluation reports are available in the appendix.

2 Problem Setting

Existing approaches to safe dynamic reconfiguration try to put the elements of the running system that are subject to change at a specific state called safe state before performing the reconfiguration operations on them. Of most relevant work in this area, quiescence [1] causes a high disruption to the running system that is not acceptable in many critical systems [7]. To address this issue, Vandewoude et al. [8] proposed the concept of tranquility, as a low disruptive alternative to quiescence:

Definition 1 (Tranquility). *A component is tranquil if: (i) it is not currently engaged in a transaction that it initiated; (ii) it will not initiate new transactions; (iii) it is not actively processing a request; and (iv) none of its adjacent components are engaged in a transaction in which both of them have already participated and might still participate in the future.*

In [9], Ma et al. show that the tranquility criterion may not guarantee safe dynamic reconfiguration of distributed transactions. Moreover, when a component is used in an infinite sequence of concurrent interleaving transactions, it is not guaranteed that it will ever reach tranquility [8]. Also, tranquility criterion is not stable by itself. Once node `N` is in a tranquil state, all interactions between `N` and its environment should be blocked to assure that the tranquil state of that node is preserved [8]. In addition, tranquility does not guarantee consistency when deleting or detaching nodes [3]. Furthermore, both notions of quiescence and tranquility assume that a valid component substitution cannot be ensured if a transaction starts with an old version of a component and finishes with the new version [10]. These issues are thoroughly discussed in [7].

---

[2] SAFe runtimE Reconfiguration.

3 Safe Reconfiguration

In many critical cases, changes should be applied as soon as the software violates a requirement, however, the consistency of changes is not guaranteed unless the affected components of the system are in a safe state. Such a precondition is not acceptable if changes in the running system are subject to very stringent time constraints to react. This section extends the notion of tranquility, and proposes *SAFER*, an approach to enable safe, low disruptive runtime evolution in distributed contexts.

3.1 The Concept of Tranquility

The notion of tranquility is a necessary criterion, but not sufficient to guarantee a safe reconfiguration [7]. In fact, the type of reconfiguration can also play an important role in the system's consistency which has not been considered by existing approaches. Therefore, we extend tranquility to guarantee a more reliable reconfiguration. To clarify the concept, the definitions of consistent reconfiguration and dependency violation are provided first:

Definition 2 (Consistent Reconfiguration). *A reconfiguration is consistent if it applies desired changes in a way that it transfers the system from a consistent configuration (before the evolution) to another consistent configuration (after the evolution). More specifically, a consistent configuration is a state of the system in which a component in a safe state can be changed without impacting both what has been already executed and what has still to be executed in active transactions.*

Definition 3 (Dependency Violation). *Modification of a component may have side effects on other components. Dependency violation is defined as the removal or modification of a certain component that leads to malfunction or failure in other component(s).*

Having these definitions in place, we define *e-tranquility*[3], an extension to tranquility as follows:

Definition 4 (E-Tranquility). *(i) node `N` is in the tranquil state; (ii) the reconfiguration does not intend to delete or unlink the node; and (iii) the change does not impose any dependency violation among the components.*

If one of these conditions is not satisfied, reaching e-tranquility delays until the transaction which `N` belongs to, is accomplished completely (either committed or rolled back). In the following section, we show that despite whether or not e-tranquility is met, leveraging SAFER, relaxes any precondition to start the evolution process. In other words, e-tranquility is only to indicate the time in which an old component can be removed safely.

[3] Extended tranquility.

3.2 SAFER: An Approach to Safe Runtime Reconfiguration

The software architecture plays a central role in achieving a safe adaptation [11]. The behavioral and structural aspects of the architecture provide useful information like component dependencies. Here, we assume the availability of such information at runtime. This is not an infeasible assumption since reflective component models, e.g., Fractal [6], not only provide such an view by introspection but also allow changing them on-the-fly by intercession.

The concept of SAFER is established based on the idea that the transactions involving a component that will be updated should be separated from new transactions as soon as the update request is issued. This is operationalized as follows:

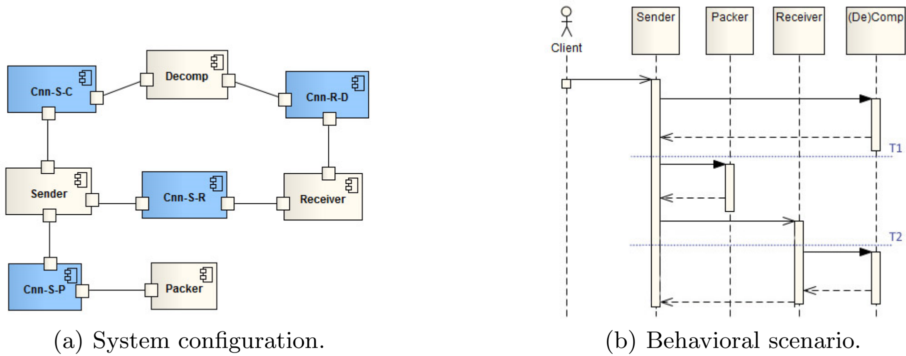
1. Whenever a component receives a change request, its new version is added to the system immediately. At this time, which is referred to as the *evolution time*, both the old and new versions of this component exist simultaneously.
2. As the second step, an event is published to dependent connectors (i.e., those which can initiate a transaction on the target component) to notify them about the start time of the evolution and the address of the new component.
3. When a connector receives a request, if the target is an evolved component `N`, the connector decides on the path to which the request should be routed, and the component which should serve the request (the old or the new version). In fact, the connector chooses the qualified component based on its knowledge about the undergoing evolution. The switching algorithm works based on the following rules: (i) if the request belongs to a transaction which has been initiated after the evolution time, it is directed to the new version of `N`; (ii) if the old version of `N` has not been used in the ongoing transaction, and reconfiguration is not resulted in a node deletion/unlinking or dependency violation, the new version is responsible for processing the request. Otherwise, the old version serves the request.
4. The switching policy continues till the old component reaches the e-tranquility. Finally, the old component will be removed completely from the system, and the completion of the evolution is notified to its dependent connectors. Accordingly, all subsequent requests will be processed by the reconfigured version.

According to SAFER, to guarantee that the old system and its related transactions will still work and that all functionalities and qualities are preserved during the reconfiguration, multiple versions of a component exist in the system until the component reaches the e-tranquility. In fact, each connector contains the intelligence necessary to manage the requests and is enriched by the information about the dependencies that exist among the components so that it can route messages to the proper version of a component.

In order to clarify SAFER, imagine a Message Delivery system as illustrated in Figure 1(a), it includes four main components which are connected by specific connectors (`Cnn-*`). Each component invoking a request to another component initiates a transaction. Each transaction as a unit of work may also need to use or

collaborate with other neighboring components by initiating new transactions (sub-transactions). The completion of a transaction depends on its execution after the termination of its sub-transactions. We assume that transactions complete in a bounded time and that the initiator of a transaction is aware of its completion [1] . Respecting this definition, a behavioral scenario of the exemplary system specified in Figure 1(b) can be therefore described as follows:

"*Whenever a client requests `Sender` to send a message, `Sender` as the root transaction invokes a request to the `(De)Compression` in order to compress the message. Next, the compressed message is sent to `Packer` to encapsulate the message with a header (including the time-stamp, message type, and decompression key) required to extract the message later. As soon as the message is prepared, it is sent to `Receiver`. On the other side, once the message is received by the `Receiver`, it asks `(De)Compression` to decompress the message to obtain the original message*".



(a) System configuration. (b) Behavioral scenario.

Fig. 1. Message Delivery System

To clarify SAFER, suppose a hypothetical situation where the `(De)Compression` component in the Message Delivery system needs to be replaced with a new version: Once an update request to `(De)Compression` is received, its new version will be added to the system. Thus, the dependent connectors of the `(De)Compression` (i.e., `Cnn-S-C` and `Cnn-R-D`) determine the component which should serve the incoming requests among a set of candidate components (the old or the new version). The coexistence of both of the versions of the `(De)Compression` continues until the old version finally reaches the e-tranquility. At this time, the old one can be removed safely from the system, and the new version of the `(De)Compression` is responsible for serving all corresponding requests. From the behavioral view of the system during the evolution, whenever `Sender` sends a request to use the compression service, since `(De)Compression` is under the evolution (two versions of this component coexist simultaneously), the request is mediated by `Cnn-S-C` to decide which version of `(De)Compression` should provide the service required by `Sender`. The target

component is chosen based on considering both the initiation time of the transaction which the request belongs to and the history of using this component in the ongoing transaction.

Regarding the definition of tranquility and the behavior of the system depicted in Figure 1(b), although `(De)Compression` is tranquil at time `T1`, due to the dependency violation, changing this component leads the system toward a failure at time `T2` where the compressed message needs to be decompressed. However, based on the e-tranquility, this component cannot be removed until the transactions involved with this component finish completely. As a result, when `Receiver` sends a decompression request at time `T2`, since the evolution is not completed yet, `Cnn-R-D` can still forward the request to the old version.

4 Related Work

Several work on software reconfiguration have been reported in literature, all of which tackle the problem from different perspectives. However, the most related approaches to deal with safe reconfiguration are considered here. These approaches fall into two major categories.

The first category of approaches is those addressing consistency through recovery. In these approaches, components can deal with the failures of an operation and recover from inconsistencies introduced during the reconfiguration. However, these approaches bring certain drawbacks: in one hand, it cannot be used in systems which do not use atomic actions. On the other hand, atomic actions are not suited to all application domains since they often result in reduced levels of concurrency in the target application. Besides, aborting transactions prior to reconfiguration is an expensive process. In contrast to the recovery approach, the second category focuses on preventing inconsistencies from occurring in the first place [8] [1] [9]. They try to put the elements of the running system that are subject to change at a specific state before performing reconfiguration operations on them. This category is of particular interest in this paper and is discussed more in the following.

Among existing approaches, a highly cited paper co-authored by Kramer and Magee [1], introduced quiescence as a reliable criterion to guarantee system consistency during the evolution. It works properly in interleaving transactions, guarantees to achieve a safe state in bounded time, and also is the only criterion which supports component removals. However, due to deactivation of all potentially related components before the reconfiguration to ensure consistency, it imposes high disruption to the running system. In order to reduce the disruption imposed by quiescence, tranquility criterion is proposed by Vandewoude et al. [8] which avoids unnecessary disturbances. Nevertheless, this criterion does not work safely in distributed transactions because of its assumption about black-box design. Moreover, there is no guarantee to achieve tranquility in bounded time in interleaving transactions [7]. In a recent work, Ma et al. [9] proposed a version-consistent approach that benefits dynamic component dependency model of the system. While it guarantees a safe dynamic reconfiguration in distributed contexts, the algorithm they propose seems to impose unnecessary processing time

to maintain dynamic dependencies when the architecture model becomes large. The time is required to reach a safe state is bounded in most approaches, while it depends on a number of transactions for version-consistency. Both tranquility and version-consistent approaches assume that the interactions with the environment should remain blocked in order to remain stable. The more assumption a specific approach is based on, a narrower application area it would be applicable. Interested readers may consult with [7].

The idea of using multi-versioning is not a new concept. Cook et al. propose a framework, HERCULES, to improve the reliability of a system by keeping existing versions of the component running and only fully removing the old component when a determination is made that the new one fully satisfies its role [12]. The new version aims to correct deficiencies that have been detected in the old version and the old version offers an example of correct behavior outside of those deficiencies. Similarly, Miki-Rakic et al. [13] encapsulate the new and old components in a wrapper component named Multi-Versioning Connector. The wrapper serves as a connector between the encapsulated component versions and the rest of the system and is responsible for propagating the generated result(s).

5 Conclusions

In this paper, we promoted connectors for enabling a safe dynamic reconfiguration by addressing the shortcomings of tranquility in distributed contexts. We demonstrated that the proposed approach, called SAFER, not only impose low disruption, but also relaxes any preconditions required to start a reconfiguration that reduces the delay within which the system is being updated. To verify the consistency of SAFER, we specified it in Alloy and applied a model checking tool to examine SAFER in different architectural configurations and also possible evolution scenarios.

We implemented SAFER in a simple example and, we compared it with other approaches based on existing information. In order to objectively compare timeliness and the disruption introduced by this approach with existing approaches, different experiments with randomly generated system configurations, different levels of workloads, and even different component models are needed. This validation is beyond the scope of this paper and we leave it for future work.

Acknowledgments. The authors would like to acknowledge Hamidreza Moradi for his help in preparing the paper. This research has been funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

Appendix

Verification Report

Dependability of runtime evolution is of major importance because any violations of consistency in running programs may lead to irreversible damages. To prevent this, reconfiguration safety should be ensured by performing appropriate (preferably formal) verification techniques. To achieve this goal, we have modeled our evolution procedure, i.e., SAFER, and its related definitions in Alloy [4] to verify whether if a consistent system is evolved using the SAFER, it still remains consistent. Due to space limitations, interested readers can refer to [5] for obtaining further details, including complete specifications of the structural, behavioral, and evolutional aspects of SAFER in Alloy

As stated earlier, a system is consistent if and only if all its transactions are consistent, and a transaction is consistent if and only if a specific version of each component is used during that transaction's lifecycle. In other words, all requests belonging to that transaction and its sub-transactions must be served by non-equivalent components.

$$
\begin{array}{l}
pred\ consistent[t : Transaction, tm : Time]\{ \\
\quad let\ exe = \{e : t.*subTransactions.actions |\ e\ in\ ExeRequest\ and\ lte[e.pre, tm]\} | \\
\qquad all\ disj\ e1, e2 : exe |\ equivalent[e1.to, e2.to] => e2.to\ in\ (e1.to).sysmetric \\
\}
\end{array}
$$

With respect to the above definition of consistent transactions, we executed the following assertion using the Alloy Analyzer to examine the reliability of SAFER for various configurations and the evolution scenarios in which we have corrective changes or node deletion/unlinkings.

$$
\begin{array}{l}
assert\ algorithmPreservesConsistency\{ \\
\quad all\ t : Transaction, en : EndNotify |\ consistent[t, en.cause.pre] => \\
\qquad\qquad\qquad\qquad\qquad\qquad consistent[t, en.post] \\
\}
\end{array}
$$

More precisely, we let the Alloy Analyzer to consider all valid configurations of totally 12 components and connectors. The invariants which we have specified through the Alloy facts forced this tool to only instantiate valid configurations of our model. In addition, before analyzing the above assertion for generated configurations, we used more other assertions based on predicates some of which are given in [5] to show that our model was valid. After executing the assertion `algorithmPreservesConsistency` for all of the instantiated configurations, the tool reported no counterexamples, or in other words, it did not find any inconsistencies. Experience has shown that if a specification has a flaw, it can usually be demonstrated by a relatively small counterexample [14].

Evaluation Report

We claim that the proposed reconfiguration approach is promising in the sense that it covers the limitations of tranquility in distributed contexts. To evaluate the applicability of SAFER in practice, it is implemented as a tool to facilitate the evolution of our running example on top of the Fractal component model [6]. After realizing the SAFER, we put its functionalities in the Fractal components membrane as controller methods to be utilized in enabling a safe reconfiguration. Interested readers are referred to [5] for details of implementation and experimental setup.

The result of this preliminary evaluation shows that in situations that a reconfiguration results in deletion of a tranquil node, the node still remains in the system to guarantee if it may, at some point in the future, participate in an ongoing transaction, even if it has not yet participated. Accordingly, the connectors could still route requests from old transactions to the deleted node. Secondly, the behavior of a component and its environmental dependencies during its execution within a transaction remains consistent. Consequently, even a temporary dependency violation does not impose any inconsistency in distributed transactions while an ongoing transaction still could use the same version of a component. In addition, the simultaneous operation of both the old and the evolved version of components enhances the availability of the system in long reconfiguration plans that include a lot of actions which will take more time to be executed and increases unavailability. Likewise, it guarantees achieving tranquility in bounded time even in the case of interleaved transactions. This is addressed by bringing a new version of the component on line to service the new top-level transactions, while the old component gradually transitions to an inactive state. Indeed, since those transactions initiated after evolution time will use new version of the component intended to be evolved, the old version would not be involved in new transactions anymore and old transactions are isolated from new interleaved ones. Furthermore, on demand possibility of using the evolved entities by new transactions increases the reliability of system in the case of critical changes like security breaches in banking services. In other words, there is no need to wait for a safe state since it is only a precondition to safely remove the old components, but not to perform the change, e.g., adding a new component.

Although the results are promising, there is still space for improvements. Having the exact component dependency model in an adaptive system is not trivial especially when the system often goes under the evolution. One way of sidestepping the overhead imposed by SAFER to keep track of transactions and their log is to collect dependencies with the mining transaction log [15]. Moreover, the overhead of performing the evolution and memory consumption of multi-version existence of the same component might impose limitations in resource-poor scenarios, especially in cases with high workloads. A remedy to this problem would be deploying changes temporary on idle resources [16].

References

1. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering 16(11), 1293–1306 (1990)
2. Crnković, I., Sentilles, S., Vulgarakis, A., Chaudron, M.: A Classification Framework for Software Component Models. IEEE Transactions on Software Engineering 37(5), 593–615 (2011)
3. Costa, C., Ali, N., Pérez, J., Carsí, J.Á., Ramos, I.: Dynamic reconfiguration of software architectures through aspects. In: Oquendo, F. (ed.) ECSA 2007. LNCS, vol. 4758, pp. 279–283. Springer, Heidelberg (2007)
4. Jackson, D.: Alloy: A lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology 11(2), 256–290 (2002)
5. Ghafari, M., Heydarnoori, A., Haghighi, H.: A safe stopping protocol to enable reliable reconfiguration for component-based distributed systems (2015), `http://home.deib.polimi.it/ghafari/SAFER.html`
6. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The FRACTAL component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems. Software: Practice and Experience 36(11-12), 1257–1284 (2006)
7. Ghafari, M., Jamshidi, P., Shahbazi, S., Haghighi, H.: Safe stopping of running component-based distributed systems: Challenges and research gaps. In: 21st IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 66–71 (2012)
8. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Transactions on Software Engineering 33(12), 856–868 (2007)
9. Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., Lu, J.: Version-consistent dynamic reconfiguration of component-based distributed systems. In: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 245–255 (2011)
10. Banno, F., Marletta, D., Pappalardo, G., Tramontana, E.: Tackling consistency issues for runtime updating distributed systems. In: IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum, pp. 1–8 (April 2010)
11. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: Framework, approaches, and styles. In: Companion of the 30th International Conference on Software Engineering, pp. 899–910 (2008)
12. Cook, J.E., Dage, J.A.: Highly reliable upgrading of components. In: 21st International Conference on Software Engineering, pp. 203–212 (1999)
13. Mikic-Rakic, M., Medvidovic, N.: Architecture-level support for software component deployment in resource constrained environments. In: IFIP/ACM Working Conference on Component Deployment, pp. 31–50 (2002)
14. Kim, J.S., Garlan, D.: Analyzing architectural styles. Journal of Systems and Software 83(7), 1216–1235 (2010)
15. Canavera, K.R., Esfahani, N., Malek, S.: Mining the execution history of a software system to infer the best time for its adaptation. In: 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 1–11 (2012)
16. Ghafari, M., Heydarnoori, A.: Partial Scalability to Ensure Reliable Dynamic Reconfiguration. In: 7th IEEE International Conference on Self-Adaptation and Self-Organizing Systems Workshops, pp. 83–88 (September 2013)