Safe Stopping of Running Component-based Distributed Systems

Challenges and Research Gaps

Mohammad Ghafari
Dept. of Computer Engineering and Information Technology, Payame Noor University, Tehran, Iran
m.ghafari@pnu.ac.ir

Pooyan Jamshidi
Lero - The Irish Software Engineering Research Centre School of Computing, Dublin City University, Ireland
pjamshidi@computing.dcu.ie

Saeed Shahbazi
Dept. of Mathematics, Faculty of Science, Khaje Nasir Toosi University of Technology, Tehran, Iran
s.shahbazi@kntu.ac.ir

Hassan Haghighi
Faculty of Electrical and Computer Engineering Shahid Beheshti University, Tehran, Iran
h_haghighi@sbu.ac.ir

***Abstract*— Continuous availability of services and low degree of disruption are two inherent necessities for mission-critical software systems. These systems could not be stopped to perform updates because disruption in their services consequent irretrievable losses. Additionally, compared to offline update, the changes should preserve the correct completion of ongoing activities. In order to place the affected elements in a safe state before dynamic changes take place, the notion of *tranquility* has been proposed to make *quiescence* criterion less disruptive and easier to obtain. Additionally, some other approaches have been proposed in order to tackle the shortcomings of these seminal proposals. However, these approaches impose some challenges to the safe dynamic reconfiguration of component-based systems. In this paper, existing challenges to preserve global consistency during runtime software reconfiguration in distributed contexts are described. The contribution of this paper is to propose a number of guidelines which can be served as agenda for future direction of research to enable a dependable safe stopping of running component-based systems.**

***Keywords*- Safe-stopping of running system; Dynamic reconfiguration; Runtime evolution; Component-based distributed system; Software architecture**

I. INTRODUCTION

Continuous availability of key functionalities and low degree of disruption are necessary for mission-critical software systems [1]. In component-based distributed systems (CBDS), components which are intrinsic parts of distributed transactions need to be replaced with newer counterparts at runtime without any disruption in services they provide. To make a change in a running system, it is usually necessary to stop a software application, apply a patch and restart it. This type of evolution is called static or offline update. However, some systems need a considerable start-up time to reach the point that they can provide their services with desired performance. In this way, the entire system services are temporarily unavailable for a period of time that might not be acceptable in an important class of software systems. In other words, in many cases, such as mission-critical systems and more concretely in the ones having long-running transactions, despite the need for updates, applications cannot be stopped to avoid possible excessive financial or human cost [1].

Compared to offline update, runtime evolution aim at evolving system without stopping and disrupting those parts of the system which are unaffected by the change [6]. Additionally, not only the system disruption during evolution should be minimized, but also it should preserve the correct completion of ongoing activities [2]. Moreover, in many critical cases, such as security threats in banking systems, the change should be undertaken on time.

In real-world systems, a variety of approaches like redundant hardware, hot pluggable devices, and system virtualization are used to provide runtime evolution. From software point of view, dynamic software updating technology addresses this challenge by updating an application while it is running. In fact, contrary to the static update, the dynamic or live update enables application updateability during its execution without restarting the whole system. However, despite significant efforts, few research systems for software-based live update have made their way into the real world, and majority of highly-available systems still rely on traditional solutions or simply do not consider live update at all [3].

The three most important issues which must be addressed in dynamic evolution are reaching a consistent application state, ensuring reliable reconfiguration, and transferring the internal state of entities which have to be replaced. The focus of this paper is on the former topic. In spite of extensive researches in dynamic evolution of component-based systems and available component models which allow software reconfiguration [4], safe reconfiguration is still an open problem [5]. A reconfiguration is safe if it preserves the whole system consistency during and after reconfiguration. The word consistency implies that a specific version of each component must being used by a transaction during its entire lifecycle. In other words, a safe update must not impact both what has been already executed and what has still to be executed in active transactions. Existing approaches which address this concern, try to put systems at specific states called safe state.

In a highly-cited paper co-authored by Kramer and Magee [6], quiescence criterion is introduced which although guarantees system consistency, it may result in significant system disruption because it blocks all potentially dependent computation during evolution. In order to reduce the disruption imposed by quiescence, Vandewoude et al. [2]

introduced the concept of tranquility as a weaker but sufficient condition for preserving system consistency during reconfiguration. However, this criterion does not work safely in distributed transactions because of the black-box design principle that they assume to be hold in each systems. Tackling to solve this global-consistency issue, Xiaoxing Ma et al. [7] proposed a version-consistent approach that guarantees safe dynamic reconfiguration in distributed contexts; however, maintaining dynamic dependencies alive imposes unnecessary processing time to the evolution process especially in real-world large scale systems.

Motivated by these concerns, in this paper, existing challenges to preserve global system consistency during the reconfiguration of component-based distributed systems are explained. We also propose number of guidelines which can be served as a future direction of research targeting the topic of enabling dependable dynamic reconfiguration especially in terms of safe stopping of the affected running entities.

The rest of this paper is organized as follows: Section II reviews the state-of-the-art approaches in the literature followed by section III that presents a motivating example to cover various reconfiguration scenarios required throughout the paper. The challenges exist in current safe stopping approaches are reviewed in section IV. Section V investigates the issues required to be addressed in real world software systems. Finally, section VI concludes the paper.

II. STATE OF THE ART APPROACHES

Runtime evolution might happen for the system element at different granularity levels. This might occur in different spectrum of software elements form coarse grained elements, such as components at the architecture level, to middle grained elements, such as component types at the specification level, to fine grained elements, such as methods at the implementation level. Runtime evolutions at the mentioned levels are called *dynamic reconfiguration*, *dynamic evolution*, and *dynamic updating*, respectively. Runtime evolution might deal with several issues to be considered as an applicable approach which preserves the integrity of the running system. The most critical issues include (i) reaching a steady state at a proper time when the system is ready to accomplish safe evolution, (ii) ensuring reliable evolution in terms of constraints which should be hold before and after change, and (iii) the transferring of state when the element ready to be evolved is stateful. These issues are corresponding to the three high-level steps in dynamic reconfiguration process as it is shown in Fig. 1. The execution states corresponding to each step of the dynamic reconfiguration process is shown to provide a better understanding of what is happening to the system execution lifecycle.

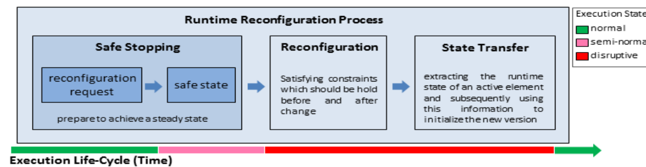

Figure 1. Dynamic Reconfiguration Process

Three prominent approaches are summarized and compared with some comparison criteria in Table 1. The comparison criteria have been derived based on the scenarios in which challenges of previous proposals have been arisen, characteristics of safe status, and formal languages that need to be adopted for specifying dynamic reconfiguration. We have chosen these approaches (as state-of-the-art approaches in safe stopping of running component-based systems) because of their extensive influence (quiescence [6]), their revolutionary improvement (tranquility [2]) and their innovation towards utilizing dynamic dependencies (version-consistent [7]). The comparison criteria have been categorized in three subject areas: key determinants of safe stopping, special situation in dynamic reconfiguration, and supplementary mechanisms.

TABLE 1. : COMPARISON OF PROMINENT SAFE STOPPING APPROACHES

Comparison Criterion		Safe Stopping Approaches		
		Quiescence	**Tranquility**	**Version consistent**
Special situation in dynamic reconfiguration	**Interleaving Transactions**	Allowed	Not allowed	Not allowed
	Condition Stability	Remain in Quiescence state when it reached this status	Interactions with environment must be blocked to remain tranquil	Interactions with environment must be blocked to remain version-consistent
	Valid component removal	Guaranteed	Not Guaranteed	Not Guaranteed
Determinants of safe stopping	**Consistency**	Globally ensured	Locally ensured	Globally ensured
	Update Latency	In bounded time	Bounded time except interleaving transactions	Workload dependent
	Disruption	High	Low	Low
	Overhead	Deactivation of related components	Checking the participation of components to specific transactions	Maintaining global dependency at runtime
	Assumption	Independent transaction	Black-box design	Global dynamic dependency
	Focus	Criterion	Criterion/Procedure	Criterion/Procedure
Supplementary mechanisms in dynamic reconfiguration	**Architectural Perspective**	Structural	Structural	Structural Behavioral
	Formalism	Graph-based	ADL	Graph-based

The other approaches lay down their contribution on different foundational assumptions. For instance, some approaches assume an especial transaction model [17], a special execution model (multi-threaded/single-threaded) [18], special component models and specific services they expose [19, 20, 21, 22], a specific architectural pattern the system is based on [23], or a special application domain such as operating system [8, 10].

The mentioned approaches tried to reduce either interruption of system's services (called disruption), especially quiescence criteria or delay (latency) to update the system. They also impose strict restrictions (such as underlying execution infrastructure) on the system to which the approaches can be applied. The more assumptions a specific approach is based on, a narrower application area it would be applicable. For instance, if an approach is based on reflection capability of a component model, it might not be applicable to the other component model that does not expose this capability. Therefore, an approach is considered a generally applicable approach if it is based on as few as possible assumptions, such as quiescence proposal that finds its way to many application areas of dynamic evolution from operating systems to real-time and embedded software applications. However, this might not be the case in some

applications because of the limited disruption they are expected.

Safe stopping of a component could be exclusively based on locally available information in that component and may derive from its interaction with the environment. In order to have a safe stopping, one could take a conservative approach that blocks the interactions of targeted component with the environment or bring the evolved version to coexist with the old one at the same time. On the other hand, one could try to gather and maintain the globally available information that are either distributed to the system or are partially exposed through underlying execution environment. The former could introduce high disruption to the system and the latter could impose high computation to the reconfiguration process or restrict it to a specific execution environment. Therefore, the level of *blocking*, level of *coexistence*, *local processing* of the globally available information and assumption on *underlying environment* are the determinants of a safe stopping approach.

III. A MOTIVATING EXAMPLE

We adopt a Message Delivery system as a simple but non-trivial example in order to describe several challenges especially in tranquility condition. As illustrated in Fig. 2, this system includes four main components, namely Sender, Receiver, (De)Compression, and Packer .

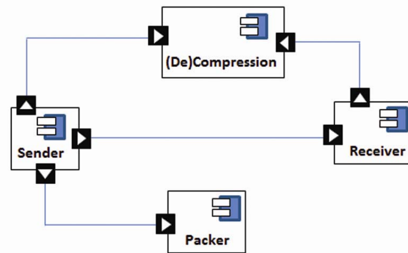

Figure 2. Message Delivery system configuration.

A behavioral scenario (transaction) is shown in Fig. 3 as follows: whenever a message is ready, the Sender sends it to the (De)Compression component in order to make its size smaller. Next, the compressed message is sent to the Packer component to be packaged. Then, as soon as the sender is identified by the Packer, a header that can consist of time-stamp, message type, decompression password, etc. is added to the message. Whenever the package is ready, it is sent to the Receiver. As soon as the Receiver receives the package, it extracts the compressed message from the package and sends it to the (De)Compression component to obtain the original message.

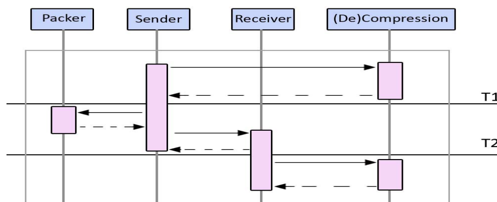

Figure 3. The sequence diagram of a sample scenario.

According to [2], a node is in a tranquil status if, by definition, does not execute code and can only be involved in an ongoing transaction when its participation in this transaction is 1) finished, 2) not yet begun, or 3) part of a sub-transaction. Although tranquility is a sufficient condition to guarantee application consistency during an update, it has some shortcomings which are investigated in the following section in detail.

IV. CHALLENGES IN SAFE STOPPING APPROACHES

The proposals mentioned in Section II introduced a major improvement towards a low disruptive alternative to the primary seminal proposal for ensuring safe dynamic reconfiguration; however, they impose some limitations as follows:

1- When a component is used in an infinite sequence of interleaving transactions, it is not guaranteed that it will ever reach tranquility [2]. An example of such case is shown in Fig. 4 which shows two interleaving transactions that are infinitely repeated. Although the (De)Compression component seems to be tranquil at time T_1 with respect to transaction X, it is still required by transaction Y after time T_1. Accordingly, since the (De)Compression component is always active in a transaction in which it still needs to participate, it can never reach tranquility. In such case, by directing Sender and Receiver components to a passive status, new transactions will not be initiated anymore, and the system will be stated in the quiescence status finally. Therefore, any systems that implements dynamic updates using the tranquility condition must implement a fallback mechanism [2].

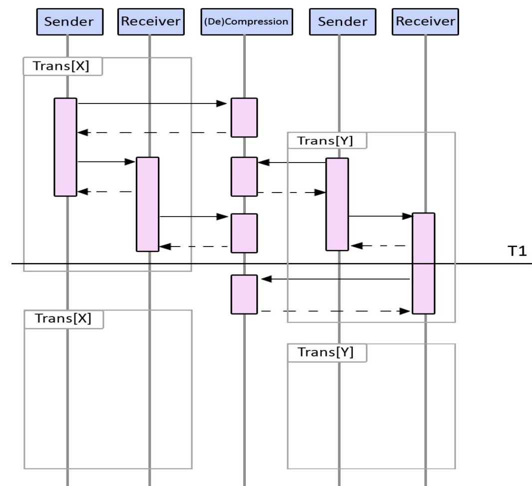

Figure 4. A scenario in which the (De)Compression component will never reach tranquility.

2- Since tranquility is based on the black-box design principle, it does not recognize distributed transactions completely. In fact, under assumption imposed by tranquility, a distributed transaction initiated by a root transaction T at node N, could only contain sub-transactions hosted by its adjacent nodes. This means that a transaction hosted by a node which is not adjacent to N, would not be part of the distributed transaction, and thus it could use any

version of components since it is an independent entity. This limitation would permit unsafe updates [7].

Furthermore, tranquility just relies on explicit direct dependencies between nodes. Therefore, if there are some implicit dependencies (mostly happening because of violating the black-box design principle) between two nodes which are not explicitly connected to each other, it may endanger safe update. Imagine that the maximum size of the compressed message in our example can be 1024 KB. Accordingly, the Packer allocates 1024 KB for encapsulating each message in the package. If a reconfiguration on the (De)Compression changes its compressed message size to 1030 KB, the length of the provided message for the Packer will be more than what it is supposed to be, and it consequences loss of data and finally ends up with a system failure. In other words, since the black box design principle is violated in this case, an implicit semantic dependency exists between the (De)Compression and the Packer: they both should care about the message compatibility [7].

3- 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. In other words, this state is not stable naturally because if a request is sent to this node, tranquility is lost. Accordingly, as soon as node N achieves tranquility, all of its interactions must be blocked to guarantee it remains in the tranquil state. However, this is not the case with quiescence, where the passivation of all nodes that can directly or indirectly initiate a new transaction on N ensures that N remains in a quiescent state until the passivated nodes are explicitly reactivated [2]. In the example, as soon as a change to the (De)Compression is required, since the passivated components could not initiate new transactions, the passivation of the Sender and Receiver guarantees achieving quiescence criterion as soon as the old transactions working with the (De)Compression are completed. In contrast, to achieve tranquility, the system should wait until it achieves this criterion naturally and then in order to remain in this state, all requests from its adjacent nodes should be blocked until the evolution is completed.

4- Tranquility does not guarantee consistency when deleting or detaching nodes [5]. The reason is as follows: a node in the tranquil status might be engaged in an ongoing transaction initiated by one of its adjacent nodes, but on which it has not participated yet. If the node is detached or removed from the system, then the ongoing transaction will fail when trying to request a service from this node; this moves the system to an inconsistent state.

5- 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 [9]. This lack of validity is due to the potential presence of symmetrical operations in the component to be substituted, as exemplified in the second challenge earlier.

In real world, not only many evolutions could be undertaken without violating an orthogonal operation, but also in many critical cases, such as security flaw, real time evolution is required. Consider the following scenarios based on our example: a breach exists in identifying the Sender by the Packer component. Any delay to fix this problem is risky, and may consequent lots of profit loss in every seconds. Therefore, while evolution to the Packer in this case does not change its wrapping format (changing its format may provide inconsistency when the package is received by the Receiver), the evolution can be undertaken as soon as this component is not actively processing a request. In another scenario, consider several simultaneous requests to Message Delivery System that may lead to QoS violations and finally denial of service. Imagine that we can optimize the compression algorithm which accordingly results in response time improvement and preventing service downtime. Although the (De)Compression component has two symmetrical operations, this evolution does not change their behavior and could be undertaken as soon as this component is idle.

V. A Research Agenda

Despite providing a transparent safe reconfiguration, a dependable approach to be applied in highly-available or mission-critical software systems should guarantee service continuousness and keeps service interruption at a minimum possible level with low performance overhead. Current approaches in software engineering have been made small steps in this direction. A safe stopping approach should satisfy some requirements depending on the situation which they might be utilized. Here are some guidelines which should be considered in order to propose an applicable safe stopping approach:

- Despite the most approaches assumption that consider the transactions completion in a bounded time, many industrial systems include some long-running transactions waiting for their completion to achieve a safe state is too risky in critical cases. In general, current approaches need some pre-conditions to achieve safe state that is not as reliable as required, because any delay in reconfiguration might raise irrecoverable consequences.
- The number of entities required to be passivated in any safe stopping approaches should be minimized to avoid any service disruption.
- Evolving a component in a running application rarely happens independently. That is, changes often result in cascade reactions in order to ensure the integrity of software architectures. Correspondingly, various components should be evolved simultaneously in many cases that are not trivial since they need coordination and special sequence by preventing any dead-locks. Suppose that component A depends on B and C depends on B. What will happen if a reconfiguration of both A and C are required since both of these components should notify B about their reconfiguration procedure. Obviously, notification from A to B puts B in a passive state. When C notifies B about its reconfiguration, B does not accept the notification because it has been passivated earlier in response to A request. Consequently, C waits for activation of B and A waits until C achieve an appropriate state for reconfiguration. As a result,

scheduling and deadlock prevention mechanisms are indispensable to have a safe approach in such situations.

- In a large-scale distributed system, the previous issue is even more complicated since it often spans multiple administrative domains and a component upgrade may require component replacements in different run-time environments. Moreover, in some cases like mechatronic systems, each discipline has its own approaches, and indeed incorporates the parts constructed by different engineering disciplines.
- Many component-based applications violate black-box design which may result some implicit dependencies between components. Therefore, some dependency check is required even if the component dependency is not reflected in the architecture model. Moreover, some external resources, e.g., memory, disk, or etc. are required to be prepared before or even during reconfiguration.
- Some component models impose several architectural constraints that might not be reflected in architecture description of the system subject to change. The safe stopping approach should take these constraints into account as well as the default architectural constraints.
- Most approaches are focused on transactional behavior. It can be interesting to investigate the systems where the behavior is continues, i.e., not transactional by default. For instance, mechatronic systems have a large number of autonomous components, exchanging information while running in parallel. Therefore, as indicated in [13], some treatment should be considered for this category of systems as well as transactional systems in order to generalize approaches.
- The approach should not depend on any reconfiguration process or mechanism. In other words, to increase the reusability of the approach in real applications, it has to be extendible and flexible enough to being utilized in varieties of reconfigurable systems from evolvable systems to self-* systems.
- In order to make the safe stopping approach applicable in different domains such as embedded systems or real-time applications (avionic, automotive, or robotic systems), they should recognize specific constraints of these domains such as the need for timely execution in a bounded time. Additionally, they need to access to real time execution environments through their underlying component model such as assigning a high priority to these reconfiguration specific tasks. Though, timeliness of the approaches depends highly on the application used, the affected components, the load on them, and their current state.
- To increase the applicability of the approach in real scenarios, different communication mechanisms should be considered. There are different mechanisms in order to connect the potentially distributed entities which raise some issues required to be taken into account in order to have a transparent reconfiguration. Particularly, the safe stopping approaches should consider the messages in transit. Some exceptions include when multiple components need to be atomically replaced [2] or an entity needs to be geographically modified [28] or there are some protocol mismatch in place [11]. For more details please consults with the cited papers.
- Any dependency to the reconfiguration manager consequents to restriction of applicability of the approach in some applications. More specifically, it should support both distributed and centralized management of adaptability.
- In order to provide a scalable approach applicable in large scale systems and to be supported by various architectural styles without many restrictions on their distribution or centralization, it should be modular and considered as a separated concern. Furthermore, this increases the applicability of the approach in legacy software systems by minimum required changes.
- Since any fault may consequent irrecoverable losses, the reconfiguration safety should be ensured by running appropriate verification, e.g., formal proof. Besides, a fallback mechanism is required to be able to switch between both the old and new versions to tolerate unanticipated faults such as hardware crashes.
- In order to enhance the feasibility of the approach in the applications which evolve frequently, it would be necessary to provide a human independent solution that also investigates both the explicit and implicit dependency relations between components automatically.
- Dynamic reconfiguration usually affects small parts of the system. If evolution becomes revolutionary and the change involves a large part of the system then the blackout time of the system and the overhead of safe stopping would outperforms the usual off-line evolution. Therefore, applicability of an approach can be evaluated before execution in order to choose the best strategy of reconfiguration. For example, if a node has many dependencies to other nodes, quiescence considerably puts the system in troubles. From another point of view, if an isolated node is responsible to serve lots of requests, achieving tranquility is almost impossible. Moreover, considering architecture topology, the components enrolled in a scenario affected by evolution, component distances and their communication protocol, some analysis can be accomplished to evaluate evolution success rate in a bounded time. Dumitras et al., [15] present an analytical framework for impact assessment to compare the risk of runtime upgrade with the risk of delaying or canceling the upgrade. Fritsch and Clarke [16] propose an admittance test for reconfigurations which determines a probability whether a reconfiguration can meet a given time bound. If the probability of meeting the constraints is high enough, the reconfiguration can be then be executed.
- There are number of recurring structural and behavioral change patterns [12] which transform a current configuration of a component-based system to the evolved configuration. As a consequence of this observation, the safe stopping approaches can be optimized by developing programmed safe status detection algorithms rather than ad-hoc ones.

It is clear that providing a safe stopping approach by taking into account all of the mentioned guidelines is not necessary and even may not be possible. However, based on the requirements of a dynamic reconfiguration process, a trade-off is required to develop an appropriate safe stopping approach. The mentioned concerns may be considered as some guidelines which future research in dynamic reconfiguration and more specifically safe stopping approaches can be taken into account.

Respecting to the current issues of safe reconfiguration, we have recently proposed a novel architectural approach enabling safe runtime evolution [14]. In order to guarantee that the old system will still work and that all functionalities and quality are preserved during reconfiguration, multi-versions of a component exists in the system until the evolution is accomplished. Each connector as a communication unit contains the intelligence necessary to manage the requests and is enriched by the information about the dependency exists between components to be able to forward messages to the proper version of a component. The provided approach makes dynamic evolution applicable in distributed contexts.

VI. Conclusion

This paper has discussed the state-of-the-art of safe stopping criteria including quiescence, tranquility and version-consistent proposals, and explained their shortcomings in details. Having identified existing challenges in this narrow track of dynamic reconfiguration research, a number of guidelines in developing an ideal approach have been proposed. The guidelines can be served as a high-level agenda for the future research in safe stopping of component-based distributed systems.

Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

References

[1] P. Oreizy, N. Medvidovic, and R. N. Taylor, “Runtime software adaptation: framework, approaches, and styles,” in Companion of the 30th international conference on Software engineering, New York, NY, USA, 2008, pp. 899–910.

[2] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D’Hondt, “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates,” Software Engineering, IEEE Transactions on, vol. 33, no. 12, pp. 856 –868, Dec. 2007.

[3] C. Giuffrida and A. S. Tanenbaum, “Cooperative update: a new model for dependable live update,” in Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades, New York, NY, USA, 2009, p. 1:1–1:6.

[4] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron, “A Classification Framework for Software Component Models,” Software Engineering, IEEE Transactions on, vol. 37, no. 5, pp. 593 – 615, Oct. 2011.

[5] C. Costa-Soria. “Dynamic evolution and reconfiguration of software architectures through aspects,” Doctoral thesis. Department of Information System and Computation, University of Politecnica De Valencia, 2011.

[6] J. Kramer and J. Magee, “The evolving philosophers problem: dynamic change management,” Software Engineering, IEEE Transactions on, vol. 16, no. 11, pp. 1293 –1306, Nov. 1990.

[7] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu, “Version-consistent dynamic reconfiguration of component-based distributed systems,” in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, New York, NY, USA, 2011, pp. 245–255.

[8] K. Makris and K. D. Ryu, “Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels,” in Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, New York, NY, USA, 2007.

[9] F. Banno, D. Marletta, G. Pappalardo, and E. Tramontana, “Tackling consistency issues for runtime updating distributed systems,” in Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, 2010, pp. 1 –8.

[10] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr, “Providing dynamic update in an operating system,” in Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, 2005, p. 32–32.

[11] C. Canal and A. Cansado, “Component reconfiguration in presence of mismatch”, Informatica 35 (2011), Pages 29–37.

[12] P. Jamshidi and C. Pahl, “Business Process and Software Architecture Model Co-Evolution Patterns” ICSE 2012 Workshop on Modelling in Software Engineering MiSE'2012.

[13] N. D. Palma, P. Laumay and L. Bellissard, “Ensuring dynamic reconfiguration consistency” In 6th International Workshop on Component-Oriented Programming (WCOP 2001), ECOOP related Workshop, pages 18–24, Budapest, Hungary, 2001.

[14] M. Ghafari, P. Jamshidi, S. Shahbazi and H. Haghighi, “An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems” in Proceedings of the 15th International ACM SIGSOFT Symposium on Component-based Software Engineering (CBSE’2012), Bertinoro, Italy, June 2012.

[15] T. Dumitras, P. Narasimhan, and E. Tilevich, “To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains,” SIGPLAN Not., vol. 45, no. 10, pp. 865–876.

[16] S. Fritsch and S. Clarke, “TimeAdapt: timely execution of dynamic software reconfigurations,” in Proceedings of the 5th Middleware doctoral symposium, New York, NY, USA, 2008, pp. 13–18.

[17] K. Moazami-Goudarzi, “Consistency preserving dynamic reconfiguration of distributed systems” Doctoral thesis. Imperial College, London, March 1999.

[18] P. Pissias and G. Coulson, “Framework for quiescence management in support of reconfigurable multi-threaded component-based systems,” Software, IET, vol. 2, no. 4, pp. 348 –361, Aug. 2008.

[19] J. Almeida, M. Wegdam, L. Pires and M. Sinderen, “An approach to dynamic reconfiguration of distributed systems based on object-middleware” in Proceedings of 19th Brazilian Symposium on Computer Networks (SBRC 2001), Santa Catarina, Brazil, May 2001.

[20] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras, “A dynamic reconfiguration service for CORBA,” in Configurable Distributed Systems, 1998. pp. 35 –42.

[21] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, “A generic component model for building systems software,” ACM Trans. Comput. Syst., vol. 26, no. 1, p. 1:1–1:42, Mar. 2008.

[22] X. Chen and M. Simons, “A Component Framework for Dynamic Reconfiguration of Distributed Systems,” in Proceedings of the IFIP/ACM Working Conference on Component Deployment, London, UK, UK, 2002, pp. 82–96.

[23] H. Gomaa and M. Hussein, “Software reconfiguration patterns for dynamic evolution of software architectures,” in Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on, 2004, pp. 79 – 88.