

ULRR

An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems

Item Type	Meetings and Proceedings
Authors	Ghafari, Mohammad;Jamshidi, Pooyan;Shahbazi, Saeed;Haghighi, Hassan
Citation	Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering;pp. 177-182
Publisher	Association for Computing Machinery
Download date	2026-04-17 22:07:00
Item License	https://creativecommons.org/licenses/by-nc-sa/1.0/
Link to Item	https://hdl.handle.net/10344/2612

An Architectural Approach to Ensure Globally Consistent Dynamic Reconfiguration of Component-Based Systems

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
Dublin, Ireland
pooyan.jamshidi
@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 Haghghi
Faculty of Electrical
and Computer Engineering
Shahid Beheshti
University
Tehran, Iran
h_haghghi
@sbu.ac.ir

ABSTRACT

One of the key issues that should be considered when addressing reliable evolution is to place a software system in a consistent status before and after change. This issue becomes more critical at runtime because it may lead to the failure on running mission-critical systems. 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. However, it only ensures consistency in applications with restrictive black-box design. In this paper, an architecture-based approach is proposed to preserve global consistency during runtime reconfiguration of component-based systems in distributed contexts. An initial evaluation through a prototypical implementation shows that this approach not only enables tranquility to be applicable for distributed transactions, but also significantly reduces required time to achieve a safe state and increases system availability during runtime evolution.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures—Languages; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Restructuring

Keywords

Dynamic reconfiguration, Component-based distributed system, Software architecture, Runtime evolution, Component model

1. INTRODUCTION

Despite extensive research in dynamic evolution of component-based systems and available component models which allow software reconfiguration [4], safe reconfiguration is still an open problem [5]. Existing approaches try to put the subject to change elements of a running system at a specific state called safe state before performing reconfiguration operations on them. A safe reconfiguration must not impact both what has been already executed and what has still to be executed in active

transactions. In other words, it should guarantee the atomicity of transactions in spite of any changes affected by evolution process. *Safe stopping criteria* are the checkable conditions that an element must satisfy to be in a safe status.

A highly-cited paper co-authored by Kramer and Magee [3], introduced *quiescence* criterion. This criterion although guarantees system consistency, it may result in significant system disruption because it blocks all potentially dependent computation during system evolution [2]. In order to reduce the disruption imposed by quiescence, *tranquility* as an alternative weaker but sufficient criterion proposed by Vandewoude et al. [2]. Tranquility is less disruptive than quiescence since it only requires blocking the requests either from or to components involved in a 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 system. In a recent work, Xiaoxing Ma et al. [7] proposed a version-consistent approach that guarantees safe dynamic reconfiguration in distributed contexts; however, it imposes unnecessary processing time to maintain dynamic dependencies when the architecture model becomes large.

Motivated by these concerns and the issues have been thoroughly discussed in [1], this paper proposes a connector-based approach which preserves global system consistency during the reconfiguration of component-based distributed systems. The key objective of this work is to promote connectors as a first class entity in software architecture description to enable safe dynamic reconfiguration with least degree of disruption and timely update by adopting previous proposals for runtime evolution.

An initial evaluation of the approach through an implementation of a component-based system by adopting a reflective component model shows that not only it enables tranquility to be applicable for distributed transactions, but also significantly reduces required time to achieve a safe state and increases system availability during dynamic reconfiguration.

This paper is organized as follows. Section 2 introduces the challenges posed by runtime evolution through an example. Our architecture-based approach for ensuring safe dynamic reconfiguration is described in Section 3 followed by an initial evaluation in Section 4. Section 5 discusses the applicability of our approach concluded by a future plan. The closest related work to our approach are studied in Section 6. Finally, Section 7 concludes the paper.

2. PROBLEM SETTING

A component-based distributed system can be described structurally as a configuration of components and connectors and behaviorally as an interaction between them. Fig. 1 shows a component-based message delivery system configuration comprising four components and their corresponding connectors as a simple but non-trivial example used throughout the paper to convey complicated concepts and help to clarify our contribution using evolution scenarios.

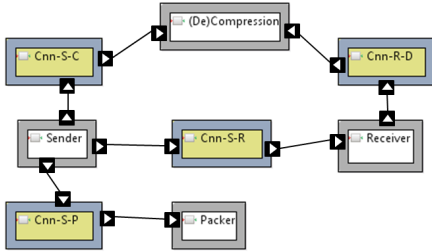


Figure 1. Message delivery system configuration.

A detailed behavioral scenario of the exemplary system is shown in Fig. 2.

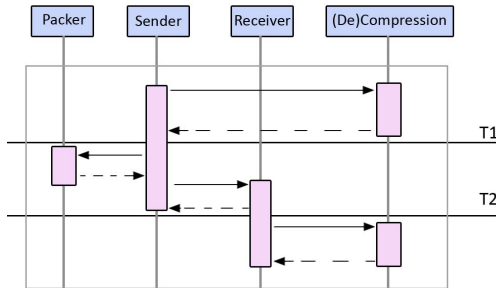


Figure 2. Message delivery system behavioral scenario.

A reliable dynamic reconfiguration transforms an existing consistent configuration of a system to another consistent configuration by conducting architecturally significant change operations. Applying such architecture change operations on system configuration may lead the state of a system to be inconsistent and behaviorally unreliable, which can also end up with a failure [5]. Because of this reason, before executing actual changes on a running system, the affected elements must be placed in a safe state, which must assure that changes on these elements do not make any inconsistencies in the system. This process must impose minimum disruption and be performed in a timely manner on the running system because we suppose that dynamic reconfigurations are needed most highly-available or mission-critical systems.

If we consider the example system, one may assume that De(Compression) component need to be updated to exploit a more efficient algorithm to reduce the size of messages. Although the new algorithm is incompatible with the old one, the other components' type do not need to be updated because all compression/decompression operations are implemented within De(Compression) component with an unchanged interface. The specification of the behavioral scenario also remains untouched; the specification-time evolution is correct since it satisfies the architecture constraints at design time. The problem is that if we update De(Compression) at runtime, we

should ensure that all messages which have been compressed can be decompressed correctly after the update has been accomplished. However, if the update were allowed to happen without any restrictions, it would be difficult to ensure it; therefore, an automatically checkable criterion should be employed to find an appropriate time to conduct the correct change [7].

According to [7] the condition must be: (i) strong enough to ensure the correctness of dynamic reconfiguration, (ii) weak enough to allow for low-disruptive and timely changes, and (iii) automatically checkable in a distributed setting.

In this area, quiescence [3] and tranquility [2] are the most extended work addressing this issue. The former poses significant disruption to the running system that is not acceptable in many critical systems [1]. On the other hand, tranquility as a low disruptive alternative to quiescence suffers from five major shortcomings has been investigated in detail in our earlier work [1].

3. DISTRIBUTED SAFE STOPPING

The approach of this paper is established based on the idea of that the component which needs to be updated should be isolated as soon as the update request is issued. Furthermore, it presents a connector driven approach to provide enough transparency during evolution. In this paper, the definition of connectors are introduced in [10]. They put the notion of connector as a first class entity in software architecture description. We intend to promote this notion in dynamic reconfiguration.

In order to minimize coupling and maximize separation of control from computation in our approach, components are can communicate through an indirect message passing. According to the taxonomy of software connectors presented in [11], our connectors provide following services:

- **Communication:** Data transfer services are the primary building blocks of component interaction. Moreover, communication protocols (e.g. procedure call, event handling, and etc.) are specified in connectors.
- **Facilitation:** Connectors mediate interaction among components without the participating components being aware of it. In our solution, each connector executes an algorithm for switching incoming request among a set of components based on its knowledge about the last evolution occurred in the system.

The notion of *serenity* is proposed as a sufficient criterion for the safety of dynamic reconfigurations (interchangeably, in this paper, runtime evolution) as follows.

Definition 1 (SERENITY). A reconfiguration to node N (interchangeably, in this paper, component, and entity) can produce a consistent configuration if all the following conditions are satisfied simultaneously:

1. Node N is in the tranquil state.
2. The reconfiguration does not intend to delete or unlink this node.
3. No semantic change is undertaken by reconfiguration to an entity which has been participated in a transaction.

In order to clarify the two critical terms in the definition 3, we provide the following definitions accordingly.

Definition 2 (CONSISTENT CONFIGURATION). *The word consistency implies that although multi version of a component might be existed within the system during its reconfiguration, a specific version of each component must be used by a transaction during its entire lifecycle and a safe update must not impact both what has been already executed and what has still to be executed in active transactions.*

Definition 3 (SEMANTIC CHANGE). *A reconfiguration to node N is considered as a semantic change if it consequents to change in either an orthogonal operation or dependency violation.*

The *serenity* indicates the time which reconfiguration process is completed, and the old entity can be removed completely from the system. Accordingly, if all three conditions above are satisfied, as soon as node N is in the tranquil state, the evolution can be performed. Otherwise, if any of these conditions are not satisfied, serenity will be extended until the transaction which N belongs to, is accomplished completely.

Considering the provided definitions, our approach to enable tranquility as a safe condition for dynamic reconfiguration in distributed transactions is proposed as the following procedure:

1. Whenever a change is required to replace component N, its new version is added to the system immediately and this time is referred as evolution time. At this time, both the old and new versions of N exist in the system simultaneously.
2. An event is published to dependent connectors of N notifying them about the start time of the evolution and the address of the new component.
3. Whenever a request is received by a connector, if the target is an evolved entity, the connector would execute an algorithm for switching the incoming request among a set of candidate components. Finally, the connector chooses the qualified component based on its knowledge about the last evolution occurred in the system. Accordingly, if the request is related to the evolved entity N, it has two options: If the request belongs to a transaction which has been initiated after evolution time, it is directed to the new version of N. Otherwise, if the transaction initiation is before the evolution time, the following mechanism will be used accordingly: If N has not been used in an ongoing transaction, and reconfiguration is not resulted in node deletion or unlinking, the new version is responsible to process the request. Otherwise, the old version serves the request.
4. At the end, as soon as the serenity criterion is satisfied, the old entity will be removed completely from the system, and the evolution accomplishment is notified to dependent connectors of N. As a result, all following requests will be processed by the reconfigured version.

The advantages of using serenity criterion as the time which completion of evolution process is notified to the affected entities is twofold: Firstly, in situations which a reconfiguration results in deletion of a tranquil node, the node must remain in the system to guarantee if it may, at some point in future, participate in an ongoing transaction, even if it has not yet participated. Accordingly, the connectors could still direct

requests from old transactions to the deleted node. Secondly, the behavior of a component and its environmental dependencies during its execution within a transaction should be consistent.

Using the provided procedure, a system has multiple states during reconfiguration as depicted in Fig. 3:

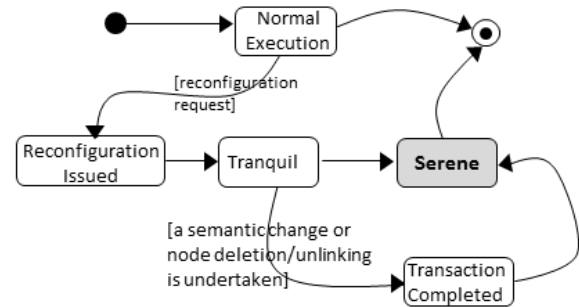


Figure 3. The system states during reconfiguration.

In order to clarify these steps, suppose a hypothetical situation where the (De)Compression component in the Message Delivery example system needs to be replaced with a new version: In a normal state of the system, as illustrated in Fig. 4(a), the (De)Compression is connected to the Sender and Receiver components by Cnn-S-C and Cnn-R-D connectors; any interactions between the (De)Compression and these components are directed by these connectors. Due to the lack of space, we only consider the interaction between the Sender and (De)Compression component in the provided sequence diagram.

Accordingly, whenever the Sender needs a message to be compressed, it sends a request to Cnn-S-C and this connector forwards the request to the (De)Compression component. Now, consider a reconfiguration process which is intended to replace (De)Compression component with its new version. As soon as the substitution request is received, the new (De)Compression component will be added to the system as depicted in Fig. 4(b). Then, the new version colored with yellow is connected to Cnn-S-C and Cnn-R-D. Accordingly, whenever a request is received by these connectors, they could transfer requests to either the old or new version of the (De)Compression component based upon situations. In other words, both versions of the (De)Compression components are running in the system simultaneously while the evolution process is progressing.

From the system behavioral point of view, whenever a request is received by a connector, if the target is still in during its evolution (or in other words, more than one version of that component exists in the system), the connector would execute an algorithm for switching incoming request among available versions of that evolved component. This algorithm determines the path to which request should be routed and the component which should serve the request.

As it is shown in the sequence diagram in Fig. 4(b), when the Sender sends a request to use a compression service from the (De)Compression component, since the (De)Compression is in the evolution state (two versions of the (De)Compression exist simultaneously), the request is mediated by Cnn-S-C to decide which version of the (De)Compression is supposed to provide the service required by the Sender. In this step, Cnn-S-C checks the initiated time of the transaction which the request belongs to. If the request belongs to a transaction which has been initiated before the evolution time, it is redirected to the old version of

the (De)Compression, provided that this component has been used earlier in this transaction; otherwise, the new version of the (De)Compression will be responsible to serve the request.

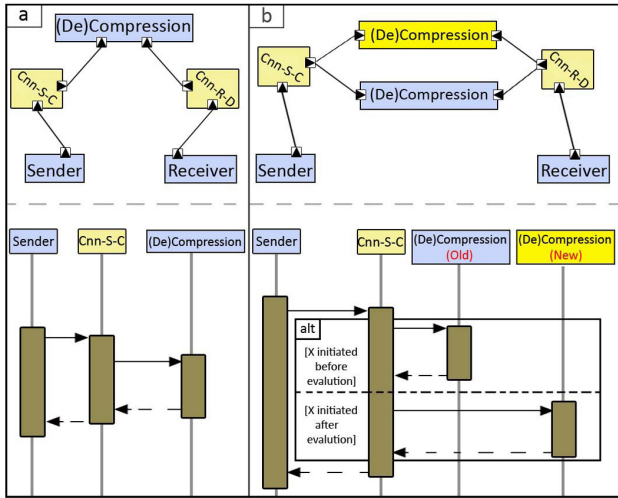


Figure 4. The example system: a) structure and behavior of the system before evolution, b) structure and behavior of the system when it receives a reconfiguration request.

Regarding the tranquility criterion, although (De)Compression is tranquil at time T_1 as depicted in Fig. 2, due to the semantic change which has been undertaken on this component, the serenity criterion is not satisfied, and this component cannot be removed from the system until its transaction finishes completely. As a result, when the Receiver sends a decompression request at time T_2 , since evolution is not completed yet, Cnn-R-D can pass its request to the old version. Finally, as soon as serenity is reached, the old version of (De)Compression is removed, and its new version is responsible to serve all corresponding requests.

4. INITIAL EVALUATION

In order to verify the practical applicability of the approach, a concrete implementation of it has been realized on top of the Fractal component model [8]. We have chosen this component model since it provides a reflective view of architecture configuration at runtime, and it supports explicit connectors that facilitate indirect message passing. We utilized FScript [9] domain specific language and FPath [9] notations to traverse the architecture model in order to program the reconfiguration logic and detect the safe stopping criteria that we need to measure. Fractal ADL [6] is also used to describe the configurations (before and after change) of the systems as case studies in this work. We utilized this because it has a built-in support for modeling various architectural elements, such as components, connectors, configuration among others and more importantly treat connector as a first class entity. Secondly, it supports runtime modification of software architecture.

After realizing the safe dynamic reconfiguration functionalities, we move them in the Fractal components membrane as controller methods. We then implement the example system on top of this component model by utilizing the extended functionalities of safe reconfiguration. The architectural configuration and behavioral model of the system with safe dynamic reconfiguration interfaces and behavior is depicted in Fig. 5 and Fig. 6 respectively.

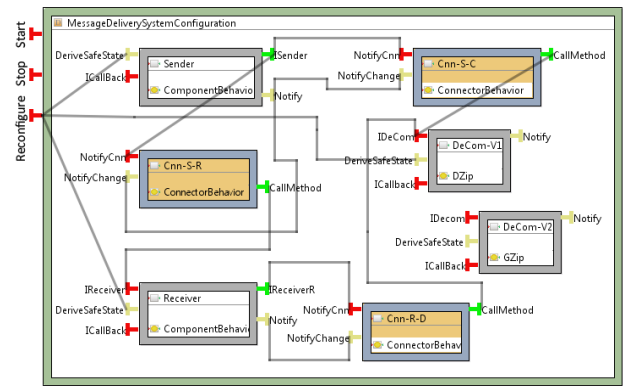


Figure 5. Message delivery system configuration with safe dynamic reconfiguration interfaces.

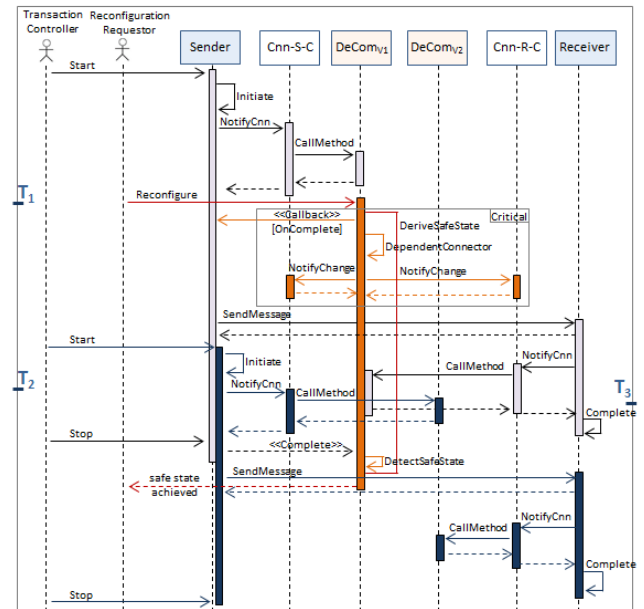


Figure 6. Message delivery system behavior with safe dynamic reconfiguration behavior.

As depicted in the sequence diagram in Fig. 6, TransactionController is responsible to start and terminate all transactions. Accordingly, it allocates the information called TransactionInfo consist of a unique Id and a time stamp as their initiation time. In order to start a transaction, the TransactionController call the Initiate method of the root initiator which acts as the start point of computation in a chain of dependent transactions. This component which is named Sender in our scenario reads a message and asks the compression component $DeCom_{V1}$ to compress it. All interactions within this system are explicitly handled by connectors. Accordingly, the compression request is first sent to Cnn-S-C and it routes the request to $DeCom_{V1}$. When the reconfiguration request is received by $DeCom_{V1}$ at time T_1 (reconfiguration time), it calls a method named DeriveSafeState to get prepared for the reconfiguration. At this time, the new version of compression component named $DeCom_{V2}$ is introduced to the system to be replaced with the $DeCom_{V1}$ as soon as it is placed in a safe state i.e. whenever there is no transaction which has been used $DeCom_{V1}$ and it may also require this component in future. Accordingly, $DeCom_{V1}$ registers itself to the root initiator as a component which should

be aware of the completion of ongoing transactions. Next, it executes a method named `DependentConnector` to find the connectors which are able to direct requests to the $DeCom_{V1}$. Accordingly, we have to find the connectors connected to the provided interfaces of $DeCom_{V1}$. Therefore, the server interfaces within $DeCom_{V1}$, i.e., `IDeCom` should be found first. Then, the connectors connected to these interfaces, i.e., `Cnn-R-C` and `Cnn-S-C` should be identified. Then, $DeCom_{V1}$ notifies `Cnn-R-C` and `Cnn-S-C` the identity of new component that is $DeCom_{V2}$ and the reconfiguration time. Once the connectors are notified, they should run a switching algorithm during their message passing between components until the reconfiguration is completed. After the message is compressed, the Sender sends it to the Receiver. In order to decompress the message, the Receiver sends a decompression request to the `Cnn-R-C`. Accordingly, the request is passed to the $DeCom_{V1}$ as it has been used earlier by this transaction. At this time, another transaction is initiated which behaves like the former until the compression request from Sender is received by `Cnn-S-C`. Since this transaction is initiated after the reconfiguration time, the request is forwarded to the $DeCom_{V2}$, new version of Compression component. As soon as the former transaction is completed at time T_3 , the Receiver as the end point of this transaction runs the `Complete` method which notifies the completion of transaction to the `TransactionController`. Correspondingly, it notifies that to the root initiator, Sender. Regarding that $DeCom_{V1}$ had registered for completion of transaction earlier; it is also informed by Sender. Accordingly, $DeCom_{V1}$ executes `DetectSafeState` to identify whether all transactions which $DeCom_{V1}$ has been attended are completed or not. If all these transactions are completed, it means $DeCom_{V1}$ is in a safe state and the reconfiguration is accomplished. Finally, the requestor is informed and $DeCom_{V1}$ can be completely replaced by $DeCom_{V2}$ safely.

5. DISCUSSION AND FUTURE PLAN

In this section, an example is presented to discuss the way by which our approach addresses tranquility shortcomings [1]. Fig. 7 shows two interleaving transactions named X and Y which are repeated infinitely. Consider that when transaction X requests a message compression by $Sender_A$, a reconfiguration is required to replace the (De)Compression component at time T_1 . Hence, the new component will be added to the system immediately, while its old version is servicing transaction X. Therefore, there is no latency between the time that the reconfiguration is requested and the time which the new component is added. At this time, all connectors connected to the old version would be connected to its new version as well to globally preserve version consistency [7] until the reconfiguration is accomplished. Therefore, whenever a request is received by a connector involved in reconfiguration, the request is forwarded to the old version of the evolved component if it has been used earlier by the ongoing transaction. On the contrary, when the old version has not been used earlier and the reconfiguration does not intend to delete or unlink the node, the new version is responsible to serve the request. Regarding to the explanations, when $Sender_B$ sends a compression request at time T_2 , since the request belongs to transaction Y initiated after the evolution time (T_1), the new version of the (De)Compression (indicated by the red timespan) serves the request. Afterwards, when transaction X requests for a decompression, since it is initiated before the reconfiguration request and (De)Compression has been used

earlier by this transaction, it is forwarded to the old version of the (De)Compression and uses the same version of (De)Compression as used earlier.

In the provided approach, since the transactions initiated after the evolution time are forwarded to use the new version of the component intended to be evolved, old transactions are isolated from new interleaved transactions and tranquility can be reached in bounded time [2]. Correspondingly, since the old version of the (De)Compression is not involved anymore in the old transactions at time T_3 , tranquility is achieved, and the old version can be removed completely from the system.

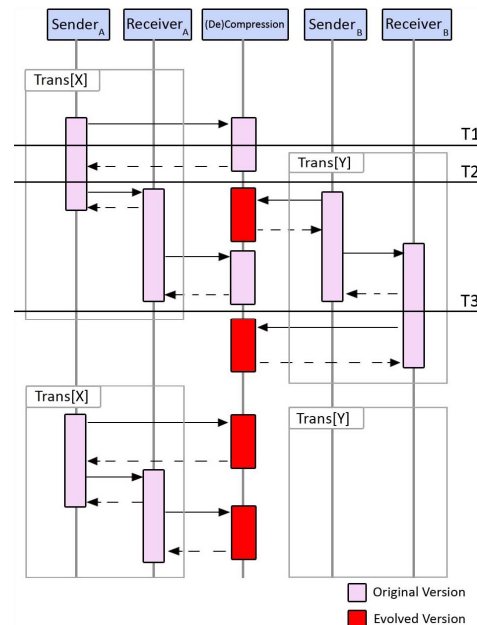


Figure 7. Application of our approach to an interleaving scenario.

We expect that our approach competes with tranquility in terms of timeliness, while the degree of disruption is independent of the system size and workloads on the component which is subject to change. The drawback of this approach seems to be twofold. Firstly, systems intended to adopt this approach should have been implemented with component models and frameworks which support connectors as first class entity. Secondly, the overhead of changing the behavior of connectors might become considerable, especially in cases with high workloads.

In order to objectively compare timeliness (the time span between receiving an update request at runtime and entering a state where the system is ready for the evolution) and disruption introduced by this approach with existing approaches (e.g., quiescence [3], tranquility [2], and version-consistent [7]), different experiments with randomly generated system configurations and different levels of workloads and even different component models or simulation settings need to be accomplished. We leave the evaluation of our proposed approach for future study.

6. RELATED WORK

Based on the topic of the work reported in this paper, approaches which their focus is on the evolution of software elements at the architecture level, and have addressed the

problem for seeking a stable state for safe dynamic reconfiguration are considered as related work. Three prominent approaches which are discussed earlier are summarized and compared with different metric as shown in Table 1. The comparison criteria have been derived based on our earlier work [1]. Moreover, characteristics of safe status and formal languages have been considered for specifying dynamic reconfiguration as well. We have chosen these approaches because of their extensive influence (quiescence [3]), 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. Further review on existing approaches is accomplished in [1].

Table 1. : The proposed approach (serenity) in comparison with state-of-the-art approaches.

Comparison Criterion	Safe Stopping Approaches				
	Quiescence	Tranquility	Version consistent	Serenity	
Special situation in dynamic reconfiguration	Interleaving Transactions	Allowed	Not allowed	Not allowed	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	Remain in stable state when it reached this status
	Valid component removal	Guaranteed	Not Guaranteed	Not Guaranteed	Guaranteed
Determinants of safe stopping	Consistency	Globally ensured	Locally ensured	Globally ensured	Globally ensured
	Update Latency	In bounded time	Bounded time except interleaving transactions	Workload dependent	No latency in viewpoint of newly initiated transactions
	Disruption	High	Low	Low	Low
	Overhead	Deactivation of related components	Checking the participation of component to specific transactions	Maintaining global dependency at runtime	Checking the initiation time of specific transaction (connector behavior specification change)
	Assumption	Independent transaction	Black-box design	Global dynamic dependency	Explicit connector support
	Focus	Criterion	Criterion/Procedure	Criterion/Procedure	Procedure
Supplementary mechanisms in dynamic reconfiguration	Architectural Perspective	Structural	Structural	Structural Behavioral	Structural Behavioral
	Formalism	Graph-based	ADL	Graph-based	ADL

There are some other approaches which lay down their contribution on different foundational assumptions. They tried to reduce either interruption of system's services introduced by previous proposals, especially quiescence criteria, or delay within which the system is being updated. They also impose strict restrictions on the system to which the approaches can be applied. The more assumption a specific approach is based on, a narrower application area it would be applicable. Therefore, an approach is considered a generally applicable approach if it is based on a least assumptions, such as quiescence proposal that found its way to many application areas of dynamic evolution form operating system to real-time and embedded software applications.

7. CONCLUSION

Connectors can potentially decouple components and facilitate distributed computing. However, their role as an appropriate means for adaptation and evolution purposes have been neglected in dynamic reconfiguration approaches. In this paper we promote connectors for realizing a safe stopping criterion which is able to address the shortcomings of tranquility in distributed contexts. The approach not only enjoys the low disruption of tranquility proposal but also showed significant reduction of required time to achieve a safe state.

The approach has been implemented using Fractal component model but it is potentially applicable to any reflective

component models with any communication protocols from synchronous to asynchronous. The distributed nature of our safe stopping functionalities which we have implemented for each component make our approach scalable that can be adopted in various environments from small scale embedded systems to highly distributed grids.

8. ACKNOWLEDGMENTS

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).

9. REFERENCES

- [1] Ghafari, M., Jamshidi, P., Shahbazi, S., Haghghi, H. 2012. Safe stopping of running component-based distributed systems: challenges and research gaps. In AROSA 2012: Proceedings of the International Conference on Adaptive and Reconfigurable Service-oriented and Component-based Applications and Architectures (Toulouse, France, June 2012).
- [2] Vandewoude, Y., Ebraert, P., Berbers, Y., and D'Hondt, T. 2007. Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Transactions on Software Engineering, (Dec. 2007), 33(12): 856-868.
- [3] Kramer, J., and Magee, J. 1990. The evolving philosophers problem: dynamic change management. IEEE Transactions on Software Engineering. 16(11):1293-1306, 1990.
- [4] Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.V., 2011. A Classification Framework for Software Component Models. Software Engineering, IEEE Transactions on 37, 593 –615.
- [5] Costa-Soria, C. 2011. Dynamic evolution and reconfiguration of software architectures through aspects. Doctoral thesis. Department of Information System and Computation, University of Politecnica De Valencia.
- [6] Leclercq, M., Özcan, A. E., Quema, V., Stefani, J.-B. 2007. Supporting heterogeneous architecture descriptions in an extensible toolset. In Proceedings of 29th International Conference on Software Engineering (Minneapolis, MN, USA, May 2007), IEEE Computer Society. 209–219, 2007.
- [7] Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., and Lu, J. 2011. Version-consistent dynamic reconfiguration of component-based distributed systems. ESEC/FSE'11 (Szeged, Hungary, Sep. 5–9, 2011).
- [8] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. 2006. The Fractal component model and its support in Java. Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems, 36(11):1257–284, Sept. 2006.
- [9] David, P.-C., Ledoux, T., Léger, M., and Coupaye, T. 2009. FPath and FScript: language support for navigation and reliable reconfiguration of Fractal architectures. Annals of Telecommunications, 64(1-2):45–63, February 2009.
- [10] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. 1995. Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering, 21(4): 314-335, Apr. 1995.
- [11] Mehta, N. R., Medvidovic, N., Phadke, S. 2000. Towards a taxonomy of software connectors. In ICSE 2000: Proceedings of the 22th International Conference on Software Engineering (Limerick, Ireland, 2000).