

Towards a Taxonomy of Dynamic Reconfiguration Approaches

Mohammad Charaf Eddin
IRIT/ Paul Sabatier University, Toulouse, France
Email: charaf@irit.fr

***Abstract*—Dynamic reconfiguration is essential part of software evolution. Several approaches to support dynamic reconfiguration have been proposed in the literature. These approaches are different in various criteria. The goal of this work is to make a comparative study between various dynamic reconfiguration approaches by exploring the features of these approaches and to classify them with respect to these features.**

***Index Terms*—reconfigurable computing systems, dynamic reconfiguration, software evolution**

I. Introduction

Different classes of software systems require the continuous availability due to safety or economical reasons (e.g., telecommunication switching systems, avionics systems, and online commercial systems). However, during the life time of a software system, many problems may appear. The problems may include programming errors, bugs, failures, and security holes. Usually, updating the software or fixing the problems may require shutting down the system and this can break the availability. Therefore, the requirement of long-availability needs different mechanisms to support online modifications without interrupting the availability.

Another class of software systems needs to adapt the changes of the user requirements or the changes of the execution environments during the running time. Consequently, such classes of systems would like to be able to make self-reconfiguration in order to adapt the unpredictable changes during the execution time. Therefore, the adaptability is another aspect of the most important motivations for supporting the online reconfigurability.

Dynamic reconfiguration [1]–[3] is a mechanism that allows the modification of a software system during the execution time without shutting it down or restarting it. Usually, a portion of software system is suspended during the reconfiguration operation whilst the unaffected parts still available. Hence, one goal of the dynamic reconfiguration approaches is to reduce the disruption time of the suspended parts in order to increase the availability rates.

Many operating systems and middleware provide some tools for loading and unloading the components (e.g., dynamic link libraries in UNIX) at running time without the need to restart the system. However, this kind of modifications is not considered as a dynamic reconfiguration modifications, because they do not have any support for consistency preservation, correctness or state transfer. Applying these mechanisms to change a running system may leave it in inconsistent state. Moreover, these mechanisms are error-prone and can break the system correctness. So, the second goal of the dynamic reconfiguration approaches is to support online modifications as well as to preserve the consistency, the structural integrity and the correctness of the modified system.

The objective of dynamic reconfiguration is to support the online changes with a minimal disruption time, and to preserve the consistency and do not violate the structural integrity of the modified system.

Several approaches are proposed in the literature to support the dynamic reconfiguration. These approaches have been researched how to preserve the consistency, the structural integrity from one hand, and how to minimize the disruption time from the other hand. We see that, it is very interesting to make a survey about a set of reconfiguration approaches and to compare their methodologies in achieving the online modification.

Dynamic reconfiguration usually performs the modifications by changing the structure of the system. The primitive operations of dynamic reconfiguration are: add, remove, replace, link, and unlink. Add or remove operations mean to add a new entity to the structure or to remove old one from it. While, link or unlink operations are used to change the interconnections between the entities.

In this paper we try to make a comparative study between reconfiguration approaches and to classify them using various criteria. The approaches difference arises from many variations like: the programming model paradigms (e.g., distributed model, procedural model) , the unit of change or the granularity of change (e.g. procedures, modules or components). Transparency, Is the approach provides transparent reconfiguration operations or not. The scalability is the approach scalable with the continuous growth of software system or not. what is the change mechanism of the approach. How the approach preserve the consistency during and after the reconfiguration.

The rest of this paper is structured as follows. First, section 2 presents many proposed criteria which will be

Manuscript received October 12, 2012; revised March 6, 2013.

doi:10.4304/jsw.8.9.2202-2207

used in the classification. Section 3 presents a survey of various selected dynamic reconfiguration approaches in different programming domains. Section 4 shows the classifications and the comparison of the surveyed approaches with the respect of the proposed criteria. Finally, we discuss the related work in section 5, and section 6 is the concluding section that summarizes the paper.

II. Exploring the Comparison Criteria

Dynamic reconfiguration approaches differ from each other in many aspects. In this section, we propose a number of various criteria in order to explore and to compare the surveyed approaches.

A. Programming Model

This criteria discusses the programming paradigm in which the dynamic reconfiguration approach works. Dynamic reconfiguration was investigated in various programming models. Some approaches deal with the object oriented programming model, other approaches deal with the distributed programming model or another model. Some approaches can span over multiple programming models. Therefore, it is interesting to classify the surveyed approaches with the respect to their programming paradigms.

B. Granularity of Change

A reconfigurable software system should be modular. This means that the structure of the system consists of several entities connected to each other. Usually, these entities represent the units of change. Dynamic reconfiguration approach may change the statements or procedures or modules or components or even subsystems. The granularity of change can be too small like a statement or can be too large like a subsystem. Each approach of dynamic reconfiguration deals with a specific unit of change. It is very recommended that the approach focuses on the coarse-grained change units like components rather than focusing on the fine-grained change units like variables and statement. Working with fine-grained level increases the complexity and the difficulties.

C. Framework

This criteria about the framework that includes the dynamic reconfiguration functionality. What are the existing frameworks that support the dynamic reconfiguration operations? These frameworks usually have the implementation of the surveyed approaches, that's to say the code libraries and the tools that provide the dynamic change management.

D. Middleware Based

The contemporary middleware provides good facilities to perform runtime modification. Dynamic reconfiguration approach can exploit these facilities to construct the change management and the reconfiguration primitive operations. Usually, these facilities perform the runtime modifications without taking in consideration the system correctness or the structural integrity. Using these facilities can leave the system in inconsistent state. Therefore, the role of dynamic reconfiguration designer is to use these facilities in safe and reliable way in order to produce a consistent system after the reconfiguration took place.

E. Consistency Preservation

The most important task of the dynamic reconfiguration approaches is to preserve the system consistency. Preserving the consistency has two challenges. The first one is how the approach preserves the global consistency and does not violate the system invariants or the structural integrity. The structural integrity property aims to satisfy a set of constraints about the system structure. For example, The binary tree structure has the following constraint: There are at most two children for each node. The second one is how the approach maintains the local consistency. The objective of local consistency is to prevent the loss of the system information during the reconfiguration process. To provide the local consistency, the approach should avoid reconfiguring the entity while it is active.

F. Script Language

Dynamic reconfiguration usually performs structural modifications on the system. The most popular primitive operations of dynamic reconfiguration are add, remove, replace, link, unlink. Add and remove can add new entity to the system or remove it, where replace operation is to replace old entity by new one link to make a connection between two entities and unlike to remove the connection. The surveyed approaches express the primitive operations in different ways. Some of them use declarative languages, the others use imperative languages. These languages may be general or restricted to a specific domain.

G. Change Mechanism

How the approach touches the running system. What is the proposed methodology to apply dynamic reconfiguration operations? Usually, the reconfiguration approaches try to put the affected entities in a safe state before reconfiguring them. Safe state comprises two things. The first one, the target entity must be frozen or inactive. That's to say the entity does not send or receive or process any information. The second one, the target entity should be isolated from the unaffected entities. The reconfiguration approaches differ from each other in the way they use to reach the safe state. Some of them use the detection way while the other drive the system to reach to the safe state. Change mechanism also should ensure that the disruption time of the whole dynamic operation does not exceed the time involved by the traditional static reconfiguration. The more the change mechanism minimizes the disruption time the more the availability maximizes.

H. Independency

This property concerns whether the approach is independent from the application context or it was developed for a specific kind of software application. The

nonindependent approaches are called adhoc approaches. Platform independent approaches are more strongly recommend than platform specific approaches.

I. Transparency

Dynamic reconfiguration transparency means that the approach is transparent from the application developer and does not require any effort from the developer to create a reconfigurable system. This quality helps to limit the complexity of building reconfigurable systems. However, nontransparent approaches require a great work from the developer to manage the reconfiguration operations. Poor transparency means that there is no explicit separation between functional concerns and reconfigurability concerns. The lack for transparency makes the dynamic reconfiguration tedious and error-prone.

J. Scalability

Dynamic reconfiguration approach should scale up as the system grows. The reconfiguration approach should be suitable for small systems as well as for large complex systems. The scalability property has a very important impact on the development time. The lack for the scalability property forces the application developer to rewrite all the reconfiguration management stuff as the system grows up.

III. Dynamic Reconfigurations Approaches Survey

In this section, we survey several approaches for supporting dynamic reconfiguration. We try to select them from various software models.

Kramer and Magee: [2] present a structural-based approach to manage the change in the transactional distributed systems. The approach uses the CONIC framework. In their approach the system consists of a set of processing nodes with directed communication links. While the transaction means an information exchange between two nodes. Transactions consist of a sequence of messages. Changes are specified in terms of the system structure (e.g., add node, remove node). A declarative language is used to specify the reconfiguration primitives. The runtime change should preserve the local consistency of affected nodes which means that there are no partially complete transactions. In order to ensure the local consistency they proposed three states for a node: active, passive, and Quiescent. Active nodes can send process and receive transactions. Passive nodes may only process and receive transactions. The quiescent state means that the node could not send or process or receive any transactions. When a runtime change is required the target node and its adjacents enter into a passive state. Then after the target node finishes the current transactions it moves to the quiescent state which means that the node is isolated and ready to accept the change. They also proved that the quiescent state is always reachable in their model of transactional systems. Their mechanism has become the de facto standard for preserving the local consistency between entities.

The first major advantage of their approach is ability to scale up as the system grows up. The second one, the approach is independent from the application context and can be used with any programming paradigm. However, it provides a poor transparency because there is a great burden from the programmer to develop the change management.

Purtilo and Hofmeister: [4] describe an approach to support dynamic reconfiguration in heterogeneous distributed applications. In this approach the system consists of several modules and bindings between them. the modules communicate to each other via the interfaces. The reconfiguration primitives here are adding and removing modules and the bindings between them. The approach use the platform POLYLITH [5] which provides both a description language and a software bus for managing the runtime activities. Their approach is strongly related to POLYLITH framework. In order to support the reconfiguration, the reconfiguration management has two operations capture and restore. Capture operation captures the current state of the executing module. The current state includes the data structure, the loop counters, file descriptors, stack variables and heap data, pc counter and other low level information. Restore operation restores the execution thread with updating the captured information. This approach does not guarantee any kind of local consistency; they have focused only on the application consistency. The absence of local consistency preservation means that the modules can be replaced even they are in an active state. Furthermore, there are poor transparency and error-prone scalability because the system developer is responsible to define the reconfiguration points explicitly in the source code.

Gupta et al: [6] propose a theoretical formal framework for modeling the online software change at the statement procedure level. The unit of change of this approach is the procedure or the program executed by a certain process. old version of a program or procedure can be replaced or updated by a new one. They describe a prototype implementation for their approach[7]. A modification shell was designed in order to accept dynamic reconfiguration commands like the command for replacing a specific routine with another one. They defined a set of conditions to ensure the validity of a change. Reconfiguration can be made at any state which satisfies these conditions. The conditions help to compute some control points where a change can be installed without any violating of validity. A change is valid if all the variables, from the control point, affected by the change are guaranteed to be redefined before any use. The consistency here is related to preserving the pointers consistency when the data is modified.

This approach only deals with the procedural programming model. The authors claimed that it can be extended to support the object oriented programming model and the distributed programming model. Therefore, scaling up this approach to large complex systems stills an open challenge. In addition, the approach is very hard,

complex, and nontransparent because it works with the low level programming like pointers, stacks.

Oreizy et al: [3] Propose an architectural-based approach to runtime software modifications. The system consists of a set of components and a set of connectors. An architectural model should be deployed with the system. This model describes the interconnections between components and connectors and their mapping to implementation modules. The deployed model is used as a basis of change. The reconfiguration operations include adding and removing components and connectors, replacing components and connectors, and changing the architectural topology. The change operations are expressed by using script language called ArchShell. Constraints are used in order to preserve the global consistency and integrity of the system. The constraints restrict the changes that violate the system integrity. To implement this approach, they have been developed a prototype using java-c2 framework[8]. The components and the connectors are implemented by using java classes. The great benefit of their approach is the separation between the runtime change management and the system functionality. Another good benefit is using the high level of abstraction by focusing on the big picture: the system components and their interconnections. So, this approach can scale up as the system grows up. However, there are many limitations in the implantation. 1) There is no support for component replacement. 2) There is no general purpose architectural-constraint mechanism. 3) Poor independency because the facilities for loading the components depend on java-c2 framework.

Bidan et al: [1] present an algorithm for making consistent dynamic reconfiguration in object oriented programming model. Their approach is built atop CORBA middleware [9]. The system consists of a set of objects that communicate over an ORB. The algorithm exploits the facilities provided by CORBA LifeCycle COS. Dynamic reconfiguration primitives include create and remove objects, link, unlink to create and destroy links, and transferlink and transferstate to transfer the pending requests on a passivated link to another activated link and the latter primitive to transfer the state from one object to another. They developed a dynamic reconfiguration manager (DRM) over CORBA to provide the reconfiguration primitives.

The algorithm adopts the quiescence approach [2] where the local consistency refers to preserving the RPC integrity. They have done a good effort for reducing the disruption time in Kramer-Magee approach. the efficiency comes from passivating the links rather than passivating the whole object. Their approach can be considered as extension of CORBA middleware. Therefore, it is application independent and it is also transparent from the developer point of view.

Cook and Dage: [10] present an architectural component-based framework for updating the system components. The framework is called HERCULES. Their approach uses coexistence policy by keeping multiple versions of a component running and contributing to the system. Reconfiguration primitives are replace and remove. The primitives are expressed by using invocations in the source code. The component replacement process has several steps. Firstly, a new version of component is created and tested, then it is installed into the running system, then the system updates its statistics about all the running versions of components and their reliability. After that, an engineer checks the whole statistics and he may execute one of the following actions: 1) remove a faulty version or unused versions 2) modify the domains of invocations.

To preserve the global consistency of the multiversion system they use two types of constraints: domain constraints and illegal domain constraints. Domain constraints specify the specific domain of correctness of the version. Illegal domain constraints specify a domain where the version should not even be executed.

The good benefit of multiversion strategy is the ability to roll back the update if the new version breaks the consistency or the functionality of the system. However, their approach provides a poor transparency for the software developer and also scaling this approach up to complex systems can result many problems due to the multiversion solution which is not an optimal solution for all classes of software systems.

IV. Classification of Dynamic Reconfiguration Approaches

In this section, we give a comparative summary of the surveyed approaches. Table I shows the programming paradigm of each approach, and the change unit and the approach framework. The survey shows that dynamic reconfiguration is very relevant in distributed systems due to their modularity. The granularity of change scales up from procedure or function to object to component.

It also shows the language which is used for expressing the reconfiguration primitives and shows whether the approach is built over middleware or not. Exploiting the facilities provided by middleware to design dynamic reconfiguration approaches can reduce the development time and make the approach more scalable and transparent. It also shows the type of consistency which is preserved by the approach local or global.

The consistency can be global or local. The global consistency concerns with the satisfaction of a set of constraints. The constraints represent the system invariants which should not be violated by the dynamic reconfiguration primitives. Local consistency is the strategy for preventing the information loss during the reconfiguration. The most famous one is the approach of Kramer-Magee [2].

Table I also compares the independency, the transparency, the scalability properties of the approaches. Providing the independency makes the approach general and not adhoc or specialized for specific applications. Supporting the transparency reduces the programming burdens and the development time. Supporting the scalability means that the approach can scale up as the system become more complex and large without the need to redevelop the change management code.

TABLE I.

COMPARATIVE SUMMARY OF THE SURVEYED APPROACHES

Approach	Programming model	Change granularity	Framework	Script	Middleware	consistency	Independency	Transparency	Scalability
Kramer	Transactional distributed systems	Process	CONIC	Declarative commands	No	Local consistency	Yes	No	Yes
Purtilo	Heterogeneous distributed applications	Module	POLYLITH	Description language	No	Application consistency	No	No	No
Gupta	Procedural programming	Procedure	Prototype	Modification shell	No	Pointers consistency	No	No	No
Oreizy	Component-based programming	Component	-	ArchShell	No	Global consistency	No	Yes	Yes
Bidan	Distributed object oriented	Object	Java-c2	DRM	CORBA	Local consistency	Yes	Yes	Yes
Cook	Component-based programming	Component	HERCULES	-	No	Global consistency	Yes	No	No

V. RELATED WORKS

In [11], the authors survey 14 approach to dynamic architectural specification. Then, they evaluate the surveyed approaches to represent the self-managing architectures.

In [12], they proposed a taxonomy of software evolution based on a large number of dimensions characterizing the mechanisms of change and the factors that influence these mechanisms. These dimensions were divided into four logical themes: temporal properties (when), object of change (where), system properties (what) and change support (how).

In [13], the authors present a large survey which has discussed the basic principles behind self-adaptive software and they propose a taxonomy of adaptation in which the questions of where, when, what, why, who, and how form the basis of this taxonomy. A landscape has been presented based on reviewing a number of disciplines related to self-adaptive software, as well as some selected research projects. A comparison between the different views of this landscape has provided a framework to identify gaps.

VI. CONCLUSIONS

In this paper, we have proposed a set of criteria in order to classify and to compare a set of dynamic reconfiguration approaches. Then we have made a survey about six different approaches for supporting dynamic reconfiguration with respecting the criteria. We can conclude with the following remarks: reconfigurable systems should be modular. It is very recommended to focus on the big picture by looking at the system as a set of components rather than diving to the statements and variables depths. Preserving the consistency is the substantial issue of dynamic reconfiguration management. Consistency can be local or global. Local consistency is used to prevent the information loss. While global consistency is used to preserve the system invariants by satisfying a set of constraints.

It is also very recommended to exploit the various facilities of the middleware in order to construct reconfigurable systems.

Finally, the following quality of service properties: independency, transparency and scalability, are very required in order to construct complex reconfigurable systems.

REFERENCES

[1] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras, "A dynamic reconfiguration service for CORBA," in *Proc. Fourth International Conference on Configurable Distributed Systems*, 1998, pp. 35–42.

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

[3] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proc. 20th international Conference on Software Engineering*, Washington, DC, USA, 1998, pp. 177–186.

[4] J. M. Purtilo and C. R. Hofmeister, "Dynamic reconfiguration of distributed programs," in *Proc. 11th International Conference on Distributed Computing Systems*, 1991, pp. 560 –571.

[5] J. M. Purtilo, "The POLYLITH software bus," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 1, pp. 151–174, Jan 1994.

[6] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 120 –131, Feb 1996.

[7] D. Gupta and P. Jalote, "On-line software version change using state transfer between processes," *Software: Practice and Experience*, vol. 23, no. 9, pp. 949–964, 1993.

[8] N. Medvidovic, P. Oreizy, and R. N. Taylor, "Reuse of off-the-shelf components in C2-style architectures," in *Proc. 19th International Conference on Software Engineering*, New York, NY, USA, 1997, pp. 692–700.

[9] J. Siegel, D. Frantz, and H. Mirsky, *et al.*, *COBRA Fundamentals and Programming*. New York, USA: John Wiley & Sons, Inc., 1996.

[10] J. E. Cook and J. A. Dage, "Highly reliable upgrading of components," in *Proc. International Conference on Software Engineering*, 1999, pp. 203–212.

[11] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications," in *Proc. 1st ACM SIGSOFT*

Workshop on Self-Managed Systems, New York, USA, 2004, pp. 28–33.
[12] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 5, pp. 309–332, 2005.
[13] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.