Theory and Applications of Mathematics & Computer Science

(ISSN 2067-2764)
http://www.uav.ro/applications/se/journal/index.php/tamcs

Theory and Applications of Mathematics & Computer Science 1 (2011) 74–79

An Overview on Software Reconfiguration

Robert Szepesi[a], Horia Ciocârlie[a,*]

[a]*Computer and Software Engineering Department "Politehnica" University of Timişoara*
V. Pârvan street 2, 300223, Timişoara Romania.

Abstract

Dynamical software reconfiguration represents a major direction in nowadays research due to its promise of providing faster solutions to ever changing problems by adding more flexibility to any given software solution at the cost of processing power, cost which given the relentless progress made by hardware manufacturers is becoming insignificant. This paper was designed as a complete overview of the software reconfiguration paradigm, looking at it from all the relevant angles - pros and cons, where to use and where not to use, challenges and solutions in implementation.

Keywords: Software reconfiguration, Reconfigurable software architecture, Design paradigms, Reconfiguration challanges.

1. Introduction

If anyone was to ever ask what was the most defining element of our times, the answer would be speed. The need for ever increasing speed is reflected in everything around us, especially in production processes, and even more so, in the software industry.

Throughout the years, the ever changing needs of the clients have prompted the software engineers to deliver software solutions at an always increasing rate, thus creating the need for the optimization of the entire creation process. First, we decided not to use chunks of code but mold those chunks into entities - classes - that could later be reused for other projects, then the classes became components, and eventually the question came what if we could reuse entire applications?

And so the idea of reconfigurable software solutions appeared - applications that could adapt themselves to a dynamic work environment without requiring the software manufacturer to change the source code in any way.

Obviously, the road from idea to actually implementing such an application is not without its bumps and that is why, throughout the length of this paper, we will discuss what exactly are these reconfigurable systems, why are they useful, where is it wise to use such a solution rather then implementing a standard application that could be patched up in time, what challenges are usually met when implementing such a solution, how they can be overcome and last, but not least, we will present a simple design that could serve for most common scenarios.

2. The What?

So what is a reconfigurable application? In short, it represents the next step on the evolutionary scale of software architectures, the key characteristic of a reconfigurable architecture being that it can adapt to the changes of either its

*Corresponding author
Email address: horia.ciocarlie@cs.upt.ro (Horia Ciocârlie)

environment, for instance the topology of a network, or the needs of its users, as long as these changes are within a predefined range, without there being any need for the system to be turned off for even a second.

However, just because a predefined range of accepted changes are allowed for the system to be configurable, it does not mean that the application in itself represents just a bundle of applications built into a single system, each configuration having assigned an internal application. The number of available configurations could, in theory, depending on the configurable parameters of the system, be infinite, but it would still require a precise set of allowed changes that the user can make - set of changes which are established through a set of parameters, each with its own set of accepted values.

In other words, a dynamically configurable software application is a generic application that can adapt its behavior, meaning data input, data processing and data presentation, at any given time , according to the values of a set of system parameters. Once having defined the system parameters, which should always be determined after carefully analyzing what is it that the system should be able to respond to, the reconfiguration process of the system can be easily defined through the change of the said parameters. However there are two ways in which the reconfiguration can be done: manual or automatic.

In the case of automatic reconfiguration, the system can determines the values of the parameters through the means of internal mechanisms that can range from sensors to simply broadcasting a message within a private network and waiting for a response. This is method of reconfiguration is used for systems that need to respond to changes occurred in the working environment of the application.

The manual reconfiguration is much simpler than the automatic one but implies the existence of a configuration panel of the system to which a (generally very limited) group of people can have access. The principle of the control panel is very simple: displays the system parameters and the allows the user to change their values, while making sure that the system can still act based on the new values.

3. The Why?

So why use dynamically configurable applications? On one hand there are systems that simply work in such dynamic environments that there is just no other solution, but for the most part, self configuration is implemented because of a number of benefits it brings to both the application per se and the software production process as well. The process of producing the software in itself is enriched by the fact that, while building a system that can later on be configured in various ways can take a longer while than building a normal application, future applications need not be created from scratch, they will simply require a proper configuration, thus saving great amount of time in the long run - this is the case of software product families, where an entire class of applications can be abstracted into a single configurable system; for more details on application families, see the following sections. When it comes to the application itself, due to the dynamic nature of the system, self configuration allows the users to perform online upgrades and even extend the application's functionalities with additional services. When talking about distributed systems, implementing self configuration capabilities into a system is almost a necessity due to the unreliable nature of a large span network which innately brings forth the need to make sure that the system survives the disconnection of one or more of the system nodes, in which case the responsibilities of one node need to be passed to others and as such, the very form of the system, as it would be represented on a graphical representation of the network, needs to be able to change while the system is running. Last, but not least, application reconfiguration becomes useful when the data that the software either produces or simply stores needs to be presented in an way that is either not unique, or simply variable in time. In these cases, the application usually allows its users to define and store, at runtime, both the ways in which the application can be presented to its users and the rules by which a certain presentation mode will be chosen over the other. Needless to say, while these are the main reasons for which reconfigurable applications are being created, they are not mutually exclusive. On the contrary, due to the ever expanding horizon of the software industry, it is expected that future generations of applications will include reconfiguration capabilities on all the levels described in this section.

4. The Where?

It should be obvious by this point that dynamic reconfiguration is not suited for any kind of application. For instance, one should not implement reconfiguration capabilities into a video game, where the software has one well

defined purpose, that does not change in time, does not operate on an unknown system configuration (every computer games has a list of system requirements attached to it, thus assuring the game developers that they know what kind of equipment will the game run on) and any other changes to the game can be done through an ulterior patch that need not be applied while the game is running. It is clear that, in such a case the application does not require reconfiguration features and implementing them would be not only pointless but also a great waste of time.

So where is it that it pays off to implement reconfiguration capabilities into the software? In general, at least one of the following questions needs to be answered with an affirmative answer before reconfiguration capabilities will be implemented into the software product:

- Is the targeted environment unstable, prone to frequent changes or simply unreliable?
- Is the way in which data is processed likely to change frequently or does data need to be processed in a different way depending on the system status?
- Is the way in which the application presents itself going to change in time?
- Will it save time in the future if the system is implemented as a general abstraction of a common problem? (to be read implemented as a family product)
- Will the system require self tuning based on self observation of execution, failure and recovery in order to provide the quality of service required?

As such, applications that usually make use of dynamic reconfiguration are either applications that run on highly dynamic equipment or applications that are part of a larger family of software products, or both. It is generally easy to determine when reconfiguration needs to be implemented due to the environment changes that the application needs to adapt to, but how does one decide if a certain application should be implemented as part of a more general family of products or an individual. In practice, the answer is always provided by the frequency with which the current problem to be solved is met. Truth be told, it is the old problem of when to create a new function/method for a piece of code versus just using that piece of code. As we all know, whenever a given piece of code, as small as it may be is expected to be used more than once, it should be factored out and implemented as a different method. If that piece of code appears in more than one places throughout the source of a component, but with some variations, the corresponding method should be parameterized so that calling the method can allow for the code to be used in all its original forms. So how do we translate this when dealing with entire applications? Let's take for instance the case of a simple management application. Most of this kind of applications use the same principle, they have a database in which they store various entities, usually products that are either being sold or produced, and an interface through which these entities are managed. The only difference between most of these applications is what the entities are, how many types of entities are there to be handled and how the data is presented to the user, and so, instead of creating a unique application each time, one would rather implement a family of applications: a template that could then be configured at run time to act as any of the applications that would've been implemented separately through the use of certain parameters.

However, it should be mentioned, that while dynamically reconfigurable software architectures are most commonly met in business and management environments, they can be successfully applied to almost all other domains. For instance, (Mun et al., 2006) describes the **Fr**actal **M**anufacturing System (FrMS) as a successful implementation of a reconfigurable software application designed specifically for the manufacturing environment.

5. The Why not?

However, the question emerges: if dynamically reconfigurable applications are so great how come the market isn't full of them yet?

The truth is that, while having reconfigurable features incorporated into an application does bring a great amount of benefit to both the software producer and user, the challenges that appear while trying to implement such a system are not few and they're also not slight.

One of the main issues is that there is no real formal method of implementing the reconfiguration of an application. In order to truly deal with reconfigurable software architectures a formal method to describe software architectures

and the changes that these need to go through has to come forth. It would be a lie to say that attempts to create such a formal method to describe the intricacies of software reconfiguration have not been made but the truth remains that most of these solutions are limited, especially when it comes to representing hierarchy and modeling context-aware systems (applications that behave differently depending on certain environment parameters, such as the geographical location of the user).

As such, without proper tools that will allow them to concretely express and document the idea upon which they intend to build, not many software engineers are inclined to take this path.

There is however hope for the future. In their paper, (Chang et al., 2008), Mao et al. propose a new formal method that could prove to fill this much needed role. Their idea is to use and extend bigraphs to describe reconfigurable software architecture, the basic idea being that bigraphs can easily survey both static and dynamic architectures through the use of graphic elements and term languages.

In addition, (Gomaa & Hussein, 2004) fully describes a few approaches for designing reconfiguration patterns, such as the *master-slave reconfiguration patterns or the centralized reconfiguration pattern*, for each of them providing modeling examples through the means of state diagrams.

Beyond the problem of having no precisely standardized way of modeling the application concept, implementing the reconfiguration capabilities of an application is not always as easy as it might appear.

Let us imagine the situation of a large distributed application, consisting of multiple processes running on different computers spread across a network. In such a case, reconfiguring the application becomes a problematic due to the concurrent nature of the system. One must always make sure that all the processes are ready to be reconfigured, which could mean that certain processes might need to be put on hold for a while or even shut down completely while the reconfiguration is performed, thus destroying the illusion of ”on the fly” reconfiguration (Mun et al., 2006). Same situation applies for when a system uses redundant servers to ensure high availability of service/data. In such cases, in order to assure the consistency of the system, one must make sure that all the system components are reconfigured at the same time, but what if one of the servers is performing an action while the server that receives the reconfiguration request is ready? Evidently one must, just like in the case of concurring processes, one must again make sure that all servers are available for reconfiguration.

A simple solution to this issue would be to simply appoint a ”reconfiguration center” somewhere within the system, perhaps even a dynamic one - the network node that received the reconfiguration request (or generated it itself after various parameter measurements) can become the reconfiguration server - which broadcasts a RECONF_ REQ message and waits until all other participants respond with a RECONF_ READY. However, while simple this solution is not always practical. What if the application is spread over a large network, is comprised of a large number of nodes, the whole request/acknowledge mechanism might generate too big of a traffic compared to how many times certain components of the system will not be ready for a reconfiguration.

A solution to this particular problem is proposed in (Whisnant et al., 2003), which suggests that dataflow dependencies among operations can be derived for any given configuration of the system if input and output signatures are created based upon the variable access patterns of the code blocks. Thus, any other proposed reconfiguration can be formally evaluated before applying it to the system, in order to determine if the new mapping of the operations to the code blocks disrupts the any dataflow dependencies currently existing within the system. Once these tests have been run on all possible configuration, using the results, either the system administrator or a part of a system itself (automated check) could use the data provided by the mechanism, which the designers named **ARMOR** (**A**daptive **R**econfigurable **M**obile **O**bjects of **R**eliability), and determine whether the system needs to use a synchronize routine before applying reconfiguration or not. Evidently, the issue risen by synchronicity is just an example, the principle presented in (Whisnant et al., 2003) can be whenever it needs to be determined if certain operations need to be executed before a configuration can be applied to the system and even to determine if the current system can actually support the configuration at hand.

6. The How?

And finally, how is the whole concept of software reconfiguration implemented in an application? While complicated in practice due to the complexity of any specific software architecture, the principle is relatively simple.

Unfortunately, changing the very blocks of an application in a transparent way at run time is a rather complicated matter, and as such, highly unlikely to be approached by many developers. One of the research directions receiving

a lot of attention regarding this matter is software dynamic translation, which represents a technology that allows the modification of an application's instructions while it is running, and strides are being made towards progress. For example, (Scott et al., 2003) describes Strata as a cross-platform infrastructure for building software dynamic translators, developed with the main goal of not only simplifying the task of initiating a new project in SDT, which the authors themselves describe as a rather difficult one, but also to promote the adoption of the SDT technology.

Imagine the components of a given application as floors of a special building. Now, the way in which the application works, is dictated by how these components are linked with one another, in other words any configuration of an application can be expressed by expressing the links between the application components. In the case of a building the floors are linked through stairs (or elevators, but, for the sake of the analogy, let's assume there are no such things as elevators). The layout of the building is expressed through the layout of the stairs. So how does a reconfigurable application work? Simple. Imagine that these stairs, like the ones presented in J.K. Rowlings books famous castle Hogwarts, can move! Every time these stairs change their location, linking different levels of the building among themselves, a person needs to follow a different path to get from point A to point B, even though points A and B have not really changed their location.

Similarly, every time a new configuration is applied to the software architecture, the links created between the software components are torn apart and new ones are created. Thus, the next time data will need to be processed by the application, like the person running through the floors of the building, it will have to follow a different path, being processed by different components and, as such, creating a different dataflow for each configuration.

Changes from one configuration to another do not have to alter the original dataflow to its very foundations, however. While it is true that the size of the components among which the links can be broken can be as small as a single variable or as huge as entire modules of the system, what it all boils down to is the fact that the way in which the basic elements of the software architecture combine changes for each configuration of the system.

But how exactly can these data and work flows actually be remodeled while the system is running? Given the extreme variety of the ways in which an application can be implemented (both in terms of coding language, each language having its own pros and cons, and in terms of methods used - for instance, an application can be implemented both by using servlets or javabeans) the ways in which the reconfiguration is just as diversified.

One of the oldest ways in which reconfiguration features were implemented in software systems is to implement the application as a set of self deployable modules that could be loaded at runtime into the main module of the application. Thus, the main module, usually representing nothing more than a figurative set of slots for the actual business modules to be inserted into, would determine once launched which are the modules are required in the current configuration of the application and then load only those modules. However, despite the amoeba-like look and feel of the application, this method of integrating reconfiguration into a software architecture proved to be inefficient due to the fact that the application is very limited when it comes to the number of valid configurations accepted. The fact is that this method represents nothing more than a fixed set of possible problems linked into a single interface, without allowing the set of problems that that system can solve ever expand without having the programmer go back into the code and creating a new module including a new solution.

The idea behind software reconfiguration is that once written it should be able to be used to solve a number of problems as large as possible, and most importantly, it should allow even its creator to be at some point surprised of what his application could be used for.

The truth is, that a reconfigurable software architecture is supposed to be nothing more, but nothing less either!, than the software equivalent of an **FPGA** (**F**ield **P**rogrammable **G**ate **A**rray) circuit, in the sense that it should obviously have a limited amount of resources available for the end user of the application, but, like an FPGA, it should allow the user to use these resources in any way it sees fit.

In fact, given nowadays technology, one could even implement a software equivalent of the FPGA that could, in a sense, offer limitless resources to its configuration administrator. The idea behind this statement is that, no matter what language the application is implemented in, one could declare dynamic classes by implementing an "entity manager" which could instantiate entities (to be read classes) with an unknown structure and eventually allowing the user, or better said a very restricted group of users, to establish these structures at runtime. This would in fact be a great example of the "two-level programs" described in (Kamin & Clausen, 2001), in which one level of the application (the entity manager part) generates the second level of the application, the actual part that the better part of the user will be interacting with.

Let's take an example to better illustrate this principle - the data management application we referred to in the

previous sections, when discussing the software product families. One could in theory declare, in code a class *CEntity* that employs an variable number of attributes implemented in a yet another variable number of classes derived from a base *CAttribute*. Thus, once the common functionalities of an entity have been implemented without caring what the attributes actually represent, a control panel could also be implemented to allow the system administrator to specify an unlimited number of entities that it would like to keep track of, each entity with its own particular structure.

Furthermore, assuming the previous principle was implemented in a well thought through manner, there is no reason why the system developers could not go a step further with the concept and allow the administrators to also define the way in which the entities interact among themselves, the way in which the data is presented, processed an so on, the whole process being based on the dynamicity brought by the Entity manager.

7. Conclusions

In conclusion, reconfigurable software architectures represent the path to be followed in order to produce high quality applications in the shortest amounts of time. While this kind of architecture does not match each and every application it can be incorporated in most data processing systems.

The idea of reconfigurable software is, however, still relatively young and, as such, challenges are met when trying to implement such an architectures due to the lack of formal methods to design and model the architectures. When trying to implement reconfiguration on large scale, on highly distributed systems, the complexity of the problem is highly increased due to accessibility, concurrency and consistency issues.

However, as more and more systems are progressing on the reconfigurable path, solutions emerge, such as those presented in the previous sections of this paper: a formal method to model the reconfigurable architecture based on bigraphs has been devised and has become quite popular, and the principle proposed by **ARMOR** could prevent most concurrency and accessibility issues when trying to apply a reconfiguration to a highly distributed system.

Last, but not least, in the last section of this paper, a design paradigm has been presented to show the true potential of the reconfigurable software architectures, which is, virtually, unlimited.

References

Chang, Z., Mao, X., & Qi, Z. (2008). Towards a formal model for reconfigurable software architectures by bigraphs. In *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)* WICSA '08 (pp. 331–334). Washington, DC, USA: IEEE Computer Society.

Gomaa, H., & Hussein, M. (2004). Software reconfiguration patterns for dynamic evolution of software architectures. In *WICSA* (pp. 79–88).

Kamin, S., & Clausen, L. (2001). Dynamically reconfigurable software components, .

Mun, J., Ryu, K., & Jung, M. (2006). Self-reconfigurable software architecture: Design and implementation. *Comput. Ind. Eng.*, *51*, 163–173.

Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J. W., & Soffa, M. L. (2003). Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* CGO '03 (pp. 36–47). Washington, DC, USA: IEEE Computer Society.

Whisnant, K., Kalbarczyk, Z. T., & Iyer, R. K. (2003). A system model for dynamically reconfigurable software. *IBM Syst. J.*, *42*, 45–59.