
Dynamic Software Architectures

Richard Torkar

Högskolan Trollhättan/Uddevalla, Sweden

richard.torkar@htu.se

Abstract: This paper covers dynamic software architectures, which is briefly mentioned in chapter 3 of “Building Reliable Component-Based Systems”. Several tools and notation languages are exposed together with a method to analyze and evaluate a dynamic software architecture. The second part of this paper shortly covers different loading techniques for components which is valid for both dynamic and static software architectures. Finally the conclusion made from the author is that dynamic software architectures are still in a very immature state just as the tools which are needed to construct these architectures.

1 Introduction

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software components [and connectors], the externally visible properties of those components [and connectors] and the relationships among them.” [1]

That sentence describes software architectures in a good way. But what are then dynamic software architectures? One could say that dynamic software architectures evolve and change during run-time - they re-configure themselves. Since they re-configure themselves we, as human beings, have problems analyzing and developing dynamic architectures. This is one of the things this paper will focus on - analyzing and designing dynamic software architectures with different tools and techniques.

The next problem one is facing, when dealing with components and dynamic architectures, is of course how these components will be dynamic with respect to their different architectures. Components can be “dynamic” in several different ways, each way with its pros and cons. The last part of this paper will focus on clarifying the five main ways a component can be dynamic.

Much has been written regarding components combined with software architectures. It is clear that a software architecture influence the behavior of components and vice versus. Many times the way the developers design an application depend on the software architecture.

2 Dynamic Software Architectures

In chapter 3, an overview regarding the development of software architectures is given. It is basic, straightforward and probably works well with static software architectures but dynamic architectures are much more complex and thus need more precise tools. Some notations, a method and a few tools will be covered next. The tools, which are explained last, will use both a notation and a type of method for creating dynamic software architectures.

Richard Torkar: Dynamic Software Architectures

Extended Report for I. Crnkovic and M. Larsson (editors), “Building Reliable Component-Based Systems”, Artech House, July 2002, ISBN 1-58053-327-2:
Chapter 3: *Architecting Component-Based Systems*, J. Bosch, J. A. Stafford

2.1 A Notation

A notation is a must when trying to illustrate and develop complex software. This is also the case when designing software architectures. Today there exist several tools for helping the designer visualize and formalize architectures and software. Rational Rose is probably the tool that has succeeded the most. But there exists several other way of visualizing an architecture. In the next pages three tools/notations will be covered. These Architectural Description Languages (ADL) are Dynamic Wright, Darwin and Koala. Unified Modeling Language (UML) will not be covered since it is strictly speaking not an ADL but a commonly used design language.

2.1.1 Dynamic Wright

Dynamic Wright [3] is an ADL that extends Wright [4]. The problem that Dynamic Wright tries to solve is simulating or notating dynamism. By using Wright as a notation language and then extend it with some additional features one can more easily describe a dynamic environment.

Take for example Figure 1, where the designer wanted to show that the client either works against a primary server and if, and only if, the primary server fails then it would start to interact with the secondary server. Here, the designer needs to symbolize, each and every point of contact (L).

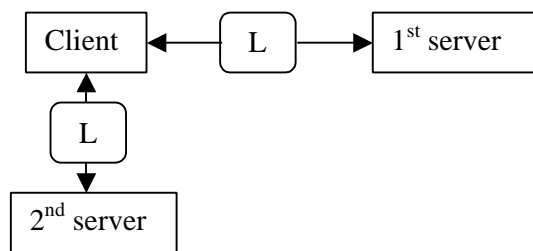


Figure 1: Static description.

The above figure is not especially clear and would take a lot of surplus text to describe and could probably lead to the designer missing some key aspects. Dynamic Wright on the other hand introduces some new notations for describing changes in the architecture. This is called a “configurator” [4] in Dynamic Wright. By adding a configurator (C) to the notation and then clarify what the control does, the designer gets a better picture of the sub system. The configurator could consist of several things, but most notably:

- When should the architecture be re-configured?
- What should cause the architecture to re-configure itself?
- How should the re-configuration be made?

Thus leading to Figure 2.

Richard Torkar: Dynamic Software Architectures

Extended Report for I. Crnkovic and M. Larsson (editors), “Building Reliable Component-Based Systems”, Artech House, July 2002, ISBN 1-58053-327-2:
Chapter 3: *Architecting Component-Based Systems*, J. Bosch, J. A. Stafford

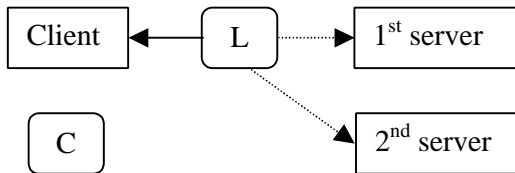


Figure 2: Dynamic description

Since Dynamic Wright is especially tailored for dynamic software architectures it makes it very suitable for these kinds of design environments.

2.1.2 Darwin

Darwin is an ADL that has been developed when working with configuration programming [16] in mind. What is so special about Darwin? Well, “[...]Darwin describes a program as a hierarchical configuration of components” [17]. This probably means that many old-time programmers find Darwin easier to use when compared to other ADLs.

Darwin as many other ADLs has both a graphical and a textual representation. One of the main differences with Darwin as suppose to other ADLs, is the rule to always specify both what a component require and provide. Figure 3 is a graphical and textual example of a component named filter taken from Jazayeri [17].

```

component filter {
  provide prev <port, int>;
  require next <port, int>;
}
  
```

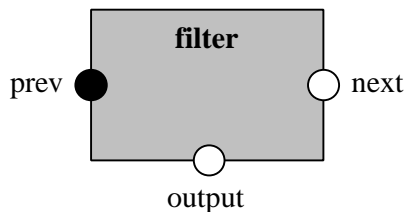


Figure 3: Component Type *filter*.

In general, a component can provide and may require several services. The names, above, are the local names for the services (i.e. next and output). Every service in Darwin is specified locally, which means that each component can be taken out from the system and tested independently. This is known as context independence.

Darwin seems to be a mature tool for designing and developing components with respect to their architectures. There seem to be much research, where Darwin is involved, in one way or another.

2.1.3 Koala

Koala is first and foremost an ADL used when developing embedded software for consumer electronics. It is basically more or less as Darwin when it comes to functionality and concepts, but Koala has a focus on resource management. Or more precisely, how not to spend resources uncontrollably.

Richard Torkar: Dynamic Software Architectures

Extended Report for I. Crnkovic and M. Larsson (editors), “Building Reliable Component-Based Systems”, Artech House, July 2002, ISBN 1-58053-327-2:
 Chapter 3: *Architecting Component-Based Systems*, J. Bosch, J. A. Stafford

Koala makes a difference when a component is used or not used. That is, a component type is a reusable component that is completely isolated and can be reused, whilst a component instance is an occurrence of a reusable component in a special context. The same thing goes for interfaces, which are divided into interface types and interface instances. An interface type is described in an interface definition language, which is in C syntax. Below is an example describing one of the interfaces in a dishwasher software.

```
Interface ProgramControl {
    void setTypeOfProgram ( DishType i );
}
```

Koala also has a graphical representation, just like Darwin as can be seen below in Figure 4.

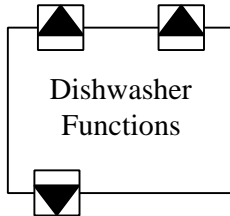


Figure 4: A component in Koala.

2.2 A Method

Let us now say that one has designed an architecture and partly implemented it. If it would have been a static architecture one could have used at least two different techniques [5,6] in evaluating and analyzing this architecture. Most notably Zhao [6] is interesting in this case. Since one usually tries to abstract the architecture, because of the complexity, slicing is an interesting approach.

Fortunately, some initial work has been done with slicing in dynamic environments [7]. By using Dynamic Software Architecture Slicing (DSAS) one could focus more on the dynamics in an architecture, or better put [7]: “[...] a dynamic software architecture slice represents the particular sequence of components and connectors of the architecture [...] with respect to a particular set of events and variable-value bindings [...]”. If one compares it to static architecture slicing, dynamic slicing generates a much smaller set of components that one needs to keep an eye on. As usual when it comes to dynamic architectures one needs to concentrate on the key functionality, simply by trying different events.

When using DSAS one also gets an algorithm that can be used to generate dynamic architecture slice. Now we have introduced a notation and a method for designing, evaluating and analyzing a dynamic software architecture. Next part will focus on a tool that can provide this help in one integrated environment.

2.3 A Tool

There is probably no tool that can help the designer to architect a dynamic software architecture from ground up. At the moment, many developers need to mix several techniques as already previously pointed out. But engineering of distributed/dynamic environments are a hassle

Richard Torkar: Dynamic Software Architectures

Extended Report for I. Crnkovic and M. Larsson (editors), “Building Reliable Component-Based Systems”, Artech House, July 2002, ISBN 1-58053-327-2:
Chapter 3: *Architecting Component-Based Systems*, J. Bosch, J. A. Stafford

nowadays. It is very complex and hard to get an overall picture of a system that has maybe hundreds or thousands sub systems.

2.3.1 PARSE-DAT

One tool has the potential to help designers with this complex task though. PARSE-DAT [8] (PARAllel Software Engineering - Design Analysis Tool) is an integrated environment that could help designers with designing and analyzing complex dynamic systems. This tool has its own notation - Dynamic PARSE-DAT, and work in three separate steps.

First, there is a pure design phase using PARSE-DAT, where one uses a particular semantics when formalizing the architecture.

Secondly, there is the translation phase, which translates the design to Milner's p-calculus [9], and finally the expressions act as input to a so-called Mobility Workbench [10]. The workbench then tries to find certain deadlocks and potential problems. PARSE-DAT is at the moment, as far as one can tell, a tool that goes far when dealing with dynamic architectures.

2.3.2 Software Architect's Assistant

Software Architect's Assistant (SAA) [18] is a tool that helps a developer design and develop a system. SAA has Darwin support built in. In the beginning SAA was designed to run under Mac OS, but since then it has been developed in Java as well. Unfortunately, SAA development seems to be discontinued.

3 Dynamic Components

Dynamic components [11] are the next thing focused on in this paper. Especially when it comes to memory footprint and CPU intensive tasks in combination with dynamic software architecture. Since dynamic software, of some sort, has played a role in traditional software engineering for a long time it becomes natural to look at dynamic software when it comes to different software architectures. In research, as well as in some commercial products, self-modifying and self-repairing code has been used [12]. Since a decade, research has also focused on "learning" computers to program themselves [13]. But first one needs to know the five basics of dynamic linking of software components.

3.1 Runtime Table Lookup

When using this technique one has to generate a table, this [table] then holds references to different external procedure calls. The symbolic references in the software then points to the addresses in the table. By doing this, the addresses in the table can be changed during run-time, since the actual starting addresses are obtained from libraries' entry. And of course, one could change the implementation of an external procedure call and not need to recompile the rest of the software.

This is one of the most basic linking procedures used in software engineering today. But it has its disadvantages, since it uses an extra memory indirection to establish the address, which programmers usually try to avoid.

Richard Torkar: Dynamic Software Architectures

Extended Report for I. Crnkovic and M. Larsson (editors), "Building Reliable Component-Based Systems", Artech House, July 2002, ISBN 1-58053-327-2:
Chapter 3: *Architecting Component-Based Systems*, J. Bosch, J. A. Stafford

3.2 Load-Time Code Modification

This method works without the extra indirection call mentioned before. In this case each and every external procedure call's address is inserted at load-time into the software. In other words, the software is modified physically. Furthermore, it means that after load time the software cannot be altered in memory thus needs to be reloaded if a new revision of a library is added to the system.

3.3 Runtime code modification

Now, an external reference does not need to be replaced during load-time. One could just as well wait until the need arises, i.e. lazy linking. This can be made by replacing every external code, during load-time, with a special call instruction and some additional information, that point to the linker.

When the external procedure call is executed in the software, it immediately passes the call to the external linker, which then executes the appropriate procedure call. By doing this, one could change the libraries in a system during runtime, and the next time the call is made it uses the new library.

3.4 Load-Time Code Generation

Not so far ago, most thought that compiling was so time and resource consuming that it was no alternative to other dynamic methods as mentioned previously. Some technologies, most notably Java [14], has proven this to be wrong. Since hardware, the last decades, has become much faster and cheaper this is no longer the case.

The Java Virtual Machine (JVM) could be said to resemble a load-time code generator. The byte code is translated to binary code by the JVM during runtime.

3.5 Full Load-Time Compilation

Full load-time compilation is the field that has not got much attention lately. Basically, what it does, is to perform a full compilation of the source code during load time. For obvious reasons, this can be quite time consuming, since the compiler not only generates code, but also performs parsing and syntax analysis of the source code.

Another huge disadvantage with full load-time compilation is that source code must always accompany the program, not only that, source code is the software being sold in this case. Even though most binary code can be decompiled rather well today, developers and commercial companies seem to feel more secure by thinking of it as impossible... In this case Free Software [15] could lead the way, but it seems unlikely.

4 Conclusions

As has been previously pointed out, software architectures have been developed in several ways lately. Some sort of notation is needed when dealing with architectures since they are complex. Even more complex is a dynamic architecture, which need some additional notation syntax, which captures the specifics regarding dynamic architectures.

Richard Torkar: Dynamic Software Architectures

Extended Report for I. Crnkovic and M. Larsson (editors), "Building Reliable Component-Based Systems", Artech House, July 2002, ISBN 1-58053-327-2:
Chapter 3: *Architecting Component-Based Systems*, J. Bosch, J. A. Stafford

Koala, a commercially used ADL, has been successful in consumer electronics. Koala, on the other hand, is an ADL suitable for embedded devices where a lack of resources is often the case. This is not what was envisioned when it comes to dynamic software architectures in this paper, thus making it not so usable for that environment.

Darwin on the other hand has all the functionality one could expect from an ADL, making it more usable in dynamic software architectures. By specifying a component the way Darwin does, one gets more or less a stand-alone component, in the sense that they can be tested and taken out of their context easily. What Darwin is missing, as far as the author can tell, is a way to enforce or add non-functional requirements on components or part of an architecture. This on the other hand, could break the context independency, something the author of this paper has not verified.

Looking at the final notation for designing software architectures, Dynamic Wright, one can see that it is not as mature as Darwin. Dynamic Wright's use of a so-called configurator makes sense, but interesting enough the developers of Dynamic Wright make no mention regarding different usage by the configurator. The developers only mention that Dynamic Wright is in no way ready for professional use. One could think of adding additional information to the configurator, which is not covered at the moment by any of the notations covered in this paper. For example non-functional requirements on a component or sub system could maybe be added to the configurator?

Different formalization languages are quite easy to find, even though they are not always suited for dynamical software architectures. On the other hand, methods for analyzing and evaluating architectures are harder to find. When it comes to dynamical software architectures, only slicing seem to be of any assistance. DSAS is the only method that really focuses on analyzing and evaluating dynamic software architectures.

When looking at different development environments, which include a notation and other tools to help the developer design a software architecture, one can easily see that they are immature. PARSE-DAT is quite new, and SAA seem to be discontinued. On the other hand are they really needed today? Several commercial actors have lately released integrated environments that can deal with static software architectures quite well. What remains to be seen is how well they can handle dynamic architectures.

When it comes to component linking, there are at the moment some methods that are more widely used, mainly because these methods are supported by some operating system or by some particular architecture. But a developer is rarely tied to one solution in today's environments. As a developer (s)he is free to use what best fit the needs of a particular project. As stated earlier, most methods can be used very well today. Mostly because today's hardware is faster, cheaper and smaller.

The technology, that most likely, will not catch on is full load-time compilation, since this requires the source code at all times. Only the Open Source world could lead the way in this field.

5 References

- [1] L. Bass, P. Clements, R. Kazman, *Software Architecture In Practise*, Addison Wesley, 1998.

Richard Torkar: *Dynamic Software Architectures*

Extended Report for I. Crnkovic and M. Larsson (editors), "Building Reliable Component-Based Systems", Artech House, July 2002, ISBN 1-58053-327-2:
Chapter 3: *Architecting Component-Based Systems*, J. Bosch, J. A. Stafford

- [2] J. Bosch, J. A. Stafford, *Architecting Component-Based Systems*, Mälardalens Högskola, 2001.
- [3] R. Allen, R. Douence & D. Garlan, *Specifying and Analyzing Dynamic Software Architectures*, Proceedings on Fundamental Approaches to Software Engineering, Lisbon, Portugal, March 1998.
- [4] R. J. Allen, *A Formal Approach to Software Architecture*, Ph.D. Thesis, TR# CMU-CS-97-144, Carnegie Mellon University, School of Computer Science, May 1997.
- [5] J. A. Stafford, D. J. Richardson & A. L. Wolf, *Aladdin: A tool for architecture-level dependence analysis of software systems*, Technical Report CU-CS-858-98, University of Colorado, Dep. of Computer Science, April 1998.
- [6] J. Zhao, *Slicing software architectures*, Technical Report 97-SE-117, Information Processing Society of Japan, pages 85-92, November 1997.
- [7] T. Kim et al, *Dynamic Software Architecture Slicing*, The University of Texas at Dallas, IEEE 0-7695-0368-3/99, 1999.
- [8] A. Liu, I. Gorton, *PARSE-DAT: An Integrated Environment for the Design and Analysis of Dynamic Software Architectures*, School of Computing Sciences, University of technology, Sydney, Australia.
- [9] R. Milner, *The Polyadic p-Calculus: a Tutorial*, Laboratory for foundations of Computer Science, Computer Science Department, University of Edingburgh, 1991.
- [10] B. Victor, *A Verification Tool for the Polyadic p-Calculus*, Licentiate Thesis, Technical report DoCS 94/50, Dept. of Computer Systems, Uppsala University, 1994. URL: <ftp://ftp.docs.uu.se/pub/mwb/>
- [11] M. Franz, *Dynamic Linking of Software Components*, IEEE Computer, 1997.
- [12] A. Ledeczi, G. Karsai & T. Bapty, *Synthesis of Self-Adaptive Software*, Institute for Software Integrated Systems 615-343-8307, Vanderbilt University, year unknown.
- [13] J. R. Koza, *Genetic Programming - On the Programming of Computers by Means of Natural Selection*, 7th ed., The MIT Press, 2000.
- [14] SUN, URL: <http://java.sun.com>, Last revised on 2002-01-04.
- [15] GNU, URL: <http://www.gnu.org>, Last revised on 2002-01-04.
- [16] J. Kramer, *Configuration programming – A framework for the development of distributable systems*, IEEE International Conference on Computer Systems and Software Engineering, California, USA, 1990.
- [17] M. Jazayeri et al, *Software Architecture for Product Families – Principles and Practice*, Addison-Wesley, 2000.
- [18] *Software Architect's Assistant*, URL: <http://www.doc.ic.ac.uk/~kn/java/saaj.html>, Last revised on 2002-01-06.

Richard Torkar: Dynamic Software Architectures

Extended Report for I. Crnkovic and M. Larsson (editors), "Building Reliable Component-Based Systems", Artech House, July 2002, ISBN 1-58053-327-2:
Chapter 3: *Architecting Component-Based Systems*, J. Bosch, J. A. Stafford